ПРОФЕССИОНАЛЬНО

Джоэл Спольски

# ДЖОЭЛ О ПРОГРАММИРОВАНИИ

И РАЗНООБРАЗНЫХ И ИНОГДА РОДСТВЕННЫХ ВОПРОСАХ, КОТОРЫЕ ДОЛЖНЫ БЫТЬ ИНТЕРЕСНЫ РАЗРАБОТЧИКАМ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ, ПРОЕКТИРОВЩИКАМ И МЕНЕДЖЕРАМ, А ТАКЖЕ ТЕМ, КОМУ ПОСЧАСТЛИВИЛОСЬ ИЛИ НЕ ПОВЕЗЛО В КАКОМ-ТО КАЧЕСТВЕ РАБОТАТЬ С НИМИ

Символ®

# JOEL ON SOFTWARE

And on Diverse and
Occasionally Related Matters
That Will Prove of Interest
to Software Developers,
Designers, and Managers,
and to Those Who,
Whether by Good Fortune
or Ill Luck, Work with Them
in Some Capacity

*Joel Spolsky*

Apress®

ПРОФЕССИОНАЛЬНО

# Джоэл о программировании

и разнообразных и иногда родственных вопросах, которые должны быть интересны разработчикам программного обеспечения, проектировщикам и менеджерам, а также тем, кому посчастливилось или не повезло в каком-то качестве работать с ними

*Джоэл Спольски*

СИМВОЛ®

*Санкт-Петербург — Москва*
*2008*

Серия «Профессионально»

Джоэл Спольски

# Джоэл о программировании

**и разнообразных и иногда родственных вопросах, которые должны быть интересны разработчикам программного обеспечения, проектировщикам и менеджерам, а также тем, кому посчастливилось или не повезло в каком-то качестве работать с ними**

Перевод С. Маккавеева


| | |
|---|---|
| Главный редактор | *А. Галунов* |
| Зав. редакцией | *Н. Макарова* |
| Научный редактор | *О. Цилюрик* |
| Редактор | *В. Овчинников* |
| Художник | *В. Гренда* |
| Верстка | *О. Макарова* |
| Корректор | *И. Губченко* |

*Спольски Дж.*

Джоэл о программировании и разнообразных и иногда родственных вопросах, которые должны быть интересны разработчикам программного обеспечения, проектировщикам и менеджерам, а также тем, кому посчастливилось или не повезло в каком-то качестве работать с ними. – Пер. с англ. – СПб.: Символ-Плюс, 2008. – 352 с., ил.

ISBN-10: 5-93286-063-4

ISBN-13: 978-5-93286-063-2


Книга представляет собой подборку эссе, опубликованных автором на его сайте *http://www.joelonsoftware.com*. Талант и глубокое проникновение в суть предмета сделали Джоэла мастером своего дела, а остроумие и едкий юмор принесли сайту скандальную известность среди программистов. Затронуты практически все вообразимые аспекты создания ПО от лучших способов устройства рабочего места программиста до лучших способов написания программного кода. Издание адресовано широкому кругу читателей – и тем, кто собирается руководить программистами, и самим программистам – как приверженцам Microsoft, так и сторонникам открытого кода.


**ISBN-10: 5-93286-063-4**

**ISBN-13: 978-5-93286-063-2**

**ISBN 1-59059-389-8 (англ)**





Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, edit@symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.
Подписано в печать 30.10.2007. Формат 70x90$^{1}/_{16}$ . Печать офсетная.
Объем 22 печ. л. Доп. тираж 2000 экз. Заказ N
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

*Моим родителям, воспитавшим меня в убеждении,*
*что все взрослые пишут книги.*

# Оглавление

# Об авторе

**Джоэл Спольски**, ветеран программной индустрии, ведет веблог «Joel on Software» (Джоэл о программировании) (*www.joelonsoftware.com*) – один из самых независимых и популярных среди программистов сайтов. Сайт Джоэла назвали «манифестом анти-Дильберта». Спольски спроектировал и написал программы, используемые миллионами людей, работал над рядом продуктов – от Microsoft Excel до пользовательского интерфейса Juno. Он основал компанию Fog Creek Software, расположенную в Нью-Йорке.

# Благодарности

Я хотел бы поблагодарить своего издателя, редактора и друга Гэри Корнелла, благодаря которому стала возможной эта книга. Сотрудники Apress всегда были любезны и добры. Apress – это редкое компьютерное издательство, в котором стремятся прежде всего заботиться об авторе, и я в восторге от совместной работы с ними.

Но еще до книги был сайт, обязанный своему существованию как бесплатному хостингу и рекламе, обеспеченным Дэйвом Винером (Dave Winer) из UserLand Software, так и его энтузиазму. Я признателен Филипу Гринспену (Philip Greenspun), который убедил меня, что знаниями надо делиться и публиковать их в Интернете, чтобы и другие могли это узнать. И Ною Тратту (Noah Tratt), вселившему в меня вдохновение, сказав однажды, что некоторые свои проповеди я должен предать бумаге.

Хочу поблагодарить своих коллег из бригады Microsoft Excel, которые на старте моей карьеры научили меня тому, как разрабатывать коммерческие приложения.

За истекшие годы мой сайт посетили миллионы людей, и примерно 10 000 из них нашли время, чтобы написать мне чудесное письмо. Эти ободряющие послания – единственная причина, по которой я продолжал писать, и хотя мне не хватит места, чтобы перечислить всех поименно, я очень ценю эти письма.

# Введение

Ты никогда не *стремился* стать менеджером. Как и большинство разработчиков программ, с которыми я знаком, ты был бы гораздо счастливее, если бы тебе позволили спокойно сидеть и писать код. Но ты лучший разработчик, и когда с Найджелом, прежним руководителем группы, произошел этот *несчастный* случай на банджи и с лэптопом, всем показалось естественным, что на его место надо выдвинуть тебя, звезду команды.

И вот теперь у тебя собственный кабинет (вместо одной клетки на двоих с вечным летним стажером), и ты должен заполнять все эти оценки эффективности труда, составляемые каждые полгода (вместо того чтобы с удовольствием портить себе зрение, глядя целый день в экран), если, конечно, ты не тратишь попусту время, разбирая причудливые требования примадонн программирования, фамильярно хлопающих по спине ребят из отдела продаж, творчески настроенных «конструкторов UI» (приглашавшихся, вообще-то, в качестве графических дизайнеров), которым нужны сверкающие кнопки OK/Cancel, способные *отражать* (простите, какое значение RGB для цвета «отражающий»?). И искать ответы на глупые вопросы старшего вице-президента, который почерпнул все свои знания о программном обеспечении из статьи в журнале, издаваемом Delta Airlines для своих пассажиров. «Почему мы используем Oracle, а не Java? Я слышал, что он более унифицирован».

Добро пожаловать в администрирование! Знаете, что я вам скажу? Управление программными проектами не имеет *никакого* отношения к программированию. Если вы до сих пор не занимались ничем, кроме написания кода, то вам предстоит открыть, что человеческие существа несколько менее предсказуемы, чем обычный процессор Intel.

Во всяком случае прежний руководитель группы, Найджел, никогда не преуспевал в этом. «Я не собираюсь стать одним из тех руководителей, которые проводят все свое время в бессмысленных совещаниях, – любил он говорить, и в этом была не только бравада. – Я думаю, что все-таки смогу 85% времени заниматься кодированием и только немножко – *администрированием*».

На самом деле Найджел *хотел* сказать другое: «У меня вообще нет *никакого* понятия о том, как руководить этим проектом, и я надеюсь, что если я просто буду продолжать кодировать так, как занимался этим до того, как меня назначили ответственным, то все как-нибудь само устроится». Оно, конечно, не устроилось, что в значительной мере и объясняет, почему Найджел прыгал на банджи вместе с IBM ThinkPad в тот злополучный день.

И Найджелу еще здорово повезло, что он выздоровел, с учетом всех обстоятельств, и сейчас он работает техническим директором небольшой компании WhatTimeIsIt.com, которую он создал вместе со своими друзьями по банджи, и у него есть всего шесть месяцев, чтобы сдать совершенно новую систему, созданную с чистого листа, и больше ему не удастся прикинуться больным.

~

Управление программными проектами не слишком хорошо изучено. Не существует степеней в управлении программными проектами, и не так много книг написано на эту тему. Кто-то из тех, кто работал над действительно удачными программными проектами, разбогател и ушел на покой разводить форель на фермах, не воспользовавшись возможностью передать накопленный ими опыт следующему поколению, а многие другие прогорели и нашли себе менее напряженную работу типа преподавания корректива английского языка хулиганам из городского гетто.

В результате многие программные проекты проваливаются, явно или скрыто, потому что ни у кого в команде нет представления о том, как должен управляться успешный программный проект. Поэтому очень многие команды так никогда и не выпускают свой продукт, или делают его слишком долго, или выпускают продукт, который никому не нужен. Но хуже всего то, что эти люди несчастны и ненавидят каждую потраченную на него минуту. Жизнь слишком коротка, чтобы ненавидеть свою работу.

Пару лет назад я опубликовал на своем сайте «тест Джоэла» – список из двенадцати характерных признаков успешно управляемых команд раз-

работчиков. В их числе такие вещи, как ведение базы данных по ошибкам, предложение поступающим на работу написать код во время собеседования и т. д. (не волнуйтесь, я подробно расскажу об этом). Поразительно, но множество людей написало мне, что их команда заработала всего лишь два или три очка из двенадцати.

Два или три!

Это почти сюрреализм. Представьте себе бригаду столяров, пытающихся делать мебель и не имеющих никакого представления о шурупах. Они употребляют исключительно гвозди и загоняют их в дерево с помощью туфель для чечетки, потому что никто не рассказывал им о существовании молотков.

Управление программными проектами требует совершенно иных навыков и приемов, нежели написание кода; это два совершенно различных и не связанных между собой поля. Написание кода так же отличается от управления проектом, как нейрохирургия от выпекания кренделей. Нет никаких оснований полагать, что у блестящего нейрохирурга, каким-то образом попавшего на производство кренделей в результате некоего разрыва в пространственно-временном континууме, окажется хоть малейшее представление о том, как делать эти самые крендели, даже если он *окончил* Гарвардский медицинский колледж. Но люди, однако, продолжают думать, что можно взять ведущего программиста и без какой-либо переподготовки перевести его в администраторы.

Так же как и вышеупомянутый нейрохирург, ты и Найджел были переведены на новую работу, в администраторы, где требуется контактировать – о, Боже! – с *людьми*, а не с компиляторами. И если тебе казалось, что современные компиляторы Java полны ошибок и непредсказуемы, то ты должен просто подождать, пока возникнет первая проблема с программистом-примадонной. Управление командами, состоящими из людей, заставит тебя смотреть на шаблоны C++ как на явно *элементарную* вещь.

Тем не менее *есть* приемы, позволяющие руководить успешными программными проектами. Это искусство шагнуло дальше гвоздей и туфель для чечетки. У нас в распоряжении молотки, отвертки и торцовочные пилы, умеющие снимать двойную фаску. В этой книге я поставил перед собой задачу познакомить читателя со всеми приемами, которые смог вспомнить, на всех уровнях – от составления графика работы руководителем команды до выработки конкурентоспособной стратегии исполнительным директором. Вы узнаете:

- Как набирать и мотивировать к работе самых лучших работников. (Это самый важный фактор успешного программного проекта.)
- Как делать реальные оценки и графики, и зачем они нужны.
- Как проектировать функции ПО и писать спецификации, которые действительно полезны для работы, а не для подшивок «один раз написал и можно не читать», из которых надстраивают перегородки в офисе.
- Как избежать распространенных ловушек в разработке ПО, и почему программисты всегда неправы, настаивая на том, что надо «все выбросить и начать сначала».
- Как организовывать команды и стимулировать их работу, и почему программистам нужны офисы с закрывающимися дверями.
- Когда следует писать собственный код с чистого листа, даже если есть почти пригодная версия, которую можно загрузить из Интернета.
- Почему всегда кажется, что программные проекты застопориваются через первую пару месяцев работы.
- Что означает стратегия ПО, и почему BeOS была обречена с первого же дня.
- А также многое другое.

Книга весьма субъективна. Ради краткости я был вынужден опустить слова типа «по моему мнению» в начале каждого предложения, потому что, как оказалось, каждое предложение в этой книге представляет мое личное мнение. И она не всеобъемлюща, но, пожалуй, сможет послужить хорошей отправной точкой.

## *А, так вы были на моем веб-сайте...*

Значительная часть содержимого этой книги впервые увидела свет в виде статей на моем веб-сайте, «Джоэл о программировании» (*www.joelonsoftware.com*), где я записываю свои мысли на протяжении последних лет. Книга, которую вы держите в руках, значительно более *связна*, как я надеюсь, чем сайт (под *связностью* я подразумеваю возможность читать, лежа в ванной, не подвергаясь риску поражения электрическим током).

Для удобства чтения мы разделили книгу на три основные секции. Первая секция посвящена разработке программного обеспечения в небольших масштабах: что надо делать, чтобы выпускать ПО, безопасное для человека. Во второй части собран ряд статей об управлении программиста-

ми и командами программистов. Третья часть составлена несколько произвольно, но в основном посвящена созданию устойчивого программного бизнеса. Вы узнаете, почему bloatware всегда побеждает, чем Ben & Jerry's отличается от Amazon, и я попытаюсь доказать, что методологии разработки программного обеспечения обычно служат признаком низкой квалификации работников.

В книге есть материал, посвященный и другим темам, но вы можете сами открыть ее и начать читать.

ЧАСТЬ ПЕРВАЯ

# Биты и байты: практика программирования

ГЛАВА ПЕРВАЯ

# Выбор языка

5 мая 2002 ГОДА, ВОСКРЕСЕНЬЕ

Почему разработчики в зависимости от конкретной задачи выбирают тот или иной язык программирования?

Иногда, если для меня важна скорость выполнения, я выбираю чистый C.

Если мне нужна программа для Windows с возможно меньшим размером дистрибутива, я часто выбираю C++ со статической компоновкой MFC.

Для GUI, который можно выполнять на Mac, в Windows и Linux, обычно выбирают Java. При этом GUI получается не самым совершенным, но вполне работоспособным.

Для быстрой разработки GUI и отточенного интерфейса пользователя я обращаюсь к Visual Basic, зная при этом, что расплачиваться придется размером дистрибутива и жесткой привязкой к Windows.

Утилиту командной строки, которая должна работать на любой UNIX-машине и не требует особой скорости, вполне можно написать на Perl.

Если выполнение должно происходить внутри веб-броузера, то фактически можно иметь дело только с JavaScript. Хранимые процедуры SQL обычно приходится писать на том диалекте SQL, который предлагает разработчик сервера, или не писать вовсе.

## *В чем смысл?*

Я очень редко выбираю язык, основываясь на синтаксисе. Да, я предпочитаю языки с {}; (такие как C/C++/C#/Java). И у меня есть свое мнение относительно того, каким должен быть «хороший» синтаксис. Но лишь ради

«точки с запятой» я не пойду на то, чтобы исполняемый модуль имел размер 20 Мбайт.

Это наводит меня на некоторые мысли относительно стратегии независимости от языка, реализованной в .NET. Она предполагает, что можно выбрать любой язык из тьмы известных, и все они будут действовать одинаково.

VB.NET и C#.NET фактически идентичны, если не считать мелких синтаксических различий. Прочие языки для участия в .NET должны поддерживать, по крайней мере, базовый набор характеристик и типов, чтобы корректно взаимодействовать с остальными. Но как можно создать в .NET утилиту командной строки UNIX? Как в .NET создать миниатюрную программу Windows EXE объемом меньше 16 Кбайт?

Похоже на то, что .NET позволяет нам «выбирать» язык, исходя из того, что как раз меньше всего нас волнует, – синтаксиса.

ГЛАВА ВТОРАЯ

# Обращаясь к основам

11 ДЕКАБРЯ 2001 ГОДА, ВТОРНИК

На своем веб-сайте я уделяю массу времени обсуждению таких «масштабных» вопросов, как сравнительные характеристики .NET и Java, стратегия развития XML, закрепление клиентов, стратегия конкурентной борьбы, проектирование программного обеспечения, архитектура и т. д. Эти темы образуют в некотором роде слоеный пирог. Верхним его слоем оказывается стратегия программного обеспечения. Ниже мы представляем себе архитектуру, например .NET, а под ней – отдельные продукты: продукты для разработки ПО, такие как Java, или платформы, такие как Windows.

Переведем взгляд еще ниже. DLL? Объекты? Функции? Нет! Ниже! В какой-то момент мы видим строки кода, написанного на том или ином языке программирования.

Но и это еще слишком высоко. Сегодня я хочу поговорить о центральном процессоре – маленьком кусочке кремния, в котором перемещаются байты. Представьте себе, что вы – начинающий программист. Забудьте все, что вы знаете о том, как пишут программы, и спуститесь на самый нижний уровень фундаментальных положений фон Неймана. Выкиньте на какое-то время из головы J2EE и подумайте на языке *байтов*.

Зачем нам это нужно? Я думаю, что некоторые крупнейшие ошибки – даже на самых верхних уровнях архитектуры – происходят из-за слабого или неверного понимания некоторых простых вещей на самых нижних уровнях. Вы построили замечательный дворец, но его фундамент никуда не годится. Вместо аккуратных бетонных плит навален булыжник. Здание выглядит прекрасно, но время от времени по совершенно непонятным причинам ванна начинает скользить по полу.

Наберитесь терпения и вместе со мной разберите одно маленькое упражнение с использованием языка программирования C.

Вспомним, как в C устроены строки: они состоят из группы байтов, за которыми следует символ null со значением 0.[1] Отсюда вытекают два очевидных следствия:

1. Невозможно узнать, где оканчивается строка (т. е. какова длина строки), без того чтобы пройти ее всю и найти в конце нулевой символ.
2. В строке не может быть нулей. Поэтому в строке C нельзя хранить произвольный двоичный объект – скажем, картинку в формате JPEG.

Почему строки C работают таким образом? Потому что в микропроцессоре PDP-7, для которого были созданы UNIX и язык программирования C, строки имели тип ASCIZ, т. е. «ASCII с Z (зеро) на конце».

Единственный ли это способ хранения строк? Нет. Но на самом деле он один из худших. Во всех программах, кроме самых тривиальных, в API, операционных системах и библиотеках классов следует избегать ASCIZ-строк, как заразы. Почему?

Для начала напишем некий вариант функции strcat, которая присоединяет одну строку к другой.

```
void strcat( char* dest, char* src )
{
   while (*dest) dest++;
   while (*dest++ = *src++);
}
```

Разберемся, как работает этот код. Сначала мы просматриваем первую строку в поиске нулевого окончания. Найдя его, мы просматриваем вторую строку, копируя каждый раз по символу в первую строку.

Такой способ обработки и конкатенации строк вполне удовлетворял Кернигана и Ритчи,[2] но с ним связаны некоторые проблемы. Приведем пример. Допустим, что есть несколько имен, которые вы хотите соединить в одну большую строку:

```
char bigString[1000];     /* Никогда не знаешь, сколько памяти выделить... */
```

---

[1] Дополнительные сведения о строках символов можно найти на *www-ee.eng.hawaii.edu/Courses/EE150/Book/chap7/subsection2.1.1.2.html*.

[2] Brian Kernighan, Dennis Ritchie «The C Programming Language», Second Edition, Prentice Hall, 1988 (Брайан Керниган и Денис Ритчи «Язык программирования C», 2 изд., Вильямс, 2005).

```
bigString[0] = '\0';
strcat(bigString,"John, ");
strcat(bigString,"Paul, ");
strcat(bigString,"George, ");
strcat(bigString,"Joel ");
```

Все работает, правда? Да. И выглядит четко и ясно.

А насколько эффективен этот код? Это самый быстрый код? А как он масштабируется? Подойдет ли он, если нужно будет добавить миллион строк?

Нет. Этот код построен на *алгоритме маляра Шлемиля*. Кто такой Шлемиль? Это малый из следующего анекдота:

> Шлемиль устроился на работу маляром и должен был наносить разметку посредине дороги. В первый день он взял бочку краски и разметил 300 метров дороги. «Неплохо! – сказал босс. – Ты быстро работаешь!» И заплатил ему денежку.
>
> На следующий день Шлемиль осилил только 150 метров. «Ну что ж, не так здорово, как вчера, но ты все равно быстро работаешь. 150 метров – это не мало», – сказал босс и заплатил ему денежку. Еще через день Шлемиль расчертил 30 метров дороги.
>
> «Всего 30 метров!» – рассвирепел босс. – Это никуда не годится. В первый день ты сделал в десять раз больше. Что случилось?»
>
> «Ничего не могу поделать, – говорит Шлемиль. – С каждым днем приходится все дальше и дальше уходить от бочки с краской».

(Для большей убедительности можно подобрать реальные цифры.)[1] Этот неважный анекдот в точности иллюстрирует механизм соединения строк в приведенном мною коде strcat. Поскольку функции приходится каждый раз просматривать всю результирующую строку и искать этот проклятый завершающий нуль, она работает гораздо медленнее, чем в действительности необходимо, и очень плохо масштабируется. Масса кода, с которым вы сталкиваетесь ежедневно, содержит эту проблему. Многие файловые системы реализованы таким образом, что при работе с ними не рекомендуется помещать в один каталог много файлов, т. к. это резко снижает производительность. Попробуйте открыть в Windows заполненную

[1] См. обсуждение арифметики на *discussfogcreek.com/techInterview/default.asp?cmd=show&ixPos t=153*.

мусорную корзину, и вы увидите, сколько для этого требуется времени, которое явно увеличивается с ростом количества файлов нелинейным образом. Где-то там спрятался алгоритм маляра Шлемиля. Всякий раз, когда чувствуется, что время должно быть линейным, а оно оказывается порядка n-квадрат, ищите Шлемиля. Часто библиотеки функций скрывают от вас это обстоятельство. Сразу и не скажешь, что вызовы strcat, помещенные в цикл или просто расположенные один под другим, вопиют «n-квадрат!», хотя именно так оно и есть в действительности.

Как с этим справиться? Некоторые хитрые С-программисты пишут собственные функции mystrcat:

```
char* mystrcat( char* dest, char* src )
{
    while (*dest) dest++;
    while (*dest++ = *src++);
    return --dest;
}
```

Что мы здесь сделали? Ценой крайне незначительных дополнительных расходов мы возвращаем указатель на *конец* новой, более длинной строки. Благодаря этому код, вызывающий данную функцию, может выполнить дописывание к строке без ее повторного просмотра следующим образом:

```
char bigString[1000];      /* Никогда не знаешь, сколько памяти выделить... */
char *p = bigString;
bigString[0] = '\0';
p = mystrcat(p,"John, ");
p = mystrcat(p,"Paul, ");
p = mystrcat(p,"George, ");
p = mystrcat(p,"Joel ");
```

Очевидно, что здесь зависимость линейная, а не квадратичная, поэтому количество конкатенируемых строк уже не пугает.

Разработчикам языка Pascal[1] эта проблема была известна, и они «решили» ее, поместив в первый байт строки счетчик байтов. Это и есть т. н. *Pascal-строки*. В них могут содержаться нули, и они не обязаны завершаться нулем.

---

[1] Разработчики PASCAL (собственно, один – Н. Вирт) создавали свое детище раньше, чем разработчики С, поэтому эта проблема была известна и тем, и другим. Оба решения имеют свои недостатки, поэтому до сих пор существует два независимых стиля представления строк: стиль С и стиль PASCAL. – *Примеч. науч. ред.*

Поскольку самое большое число, которое можно записать в байт, – это 255, Pascal-строки не могут быть длиннее 255 байт, но т. к. они не завершаются нулем, то занимают столько же памяти, сколько строки ASCIZ. Замечательно в Pascal-строках то, что не нужно организовывать цикл для того, чтобы вычислить длину строки. Узнать длину строки в Pascal можно с помощью одной команды ассемблера, не выполняя цикла. Это гораздо быстрее.

Раньше в операционных системах на Макинтошах Pascal-строки использовались повсеместно. Многие C-программисты выбирали Pascal-строки на других платформах для увеличения скорости. Excel внутренне использует Pascal-строки, из-за чего во многих случаях длина строки в Excel ограничена 255 байтами, но это и одна из причин, по которым Excel обладает столь высоким быстродействием.

Раньше литерал Pascal-строки внедрялся в код C так:

```
char* str = "\006Hello!";
```

Да, надо было сосчитать байты самому и записать это число в первый байт строки. Ленивые программисты поступали так, замедляя выполнение своих программ:

```
char* str = "*Hello!";
str[0] = strlen(str) - 1;
```

Обратите внимание, что в данном случае получается строка с нулевым окончанием (это делает компилятор) и одновременно Pascal-строка. Когда мне приходится говорить о таких строках, я обычно вспоминаю какое-нибудь короткое ругательство, ведь это проще, чем сказать *Pascal-строки, оканчивающиеся символом null*, но мы с вами находимся в приличном месте, и поэтому придется примириться с длинным названием.

Я опустил важный вопрос. Вспомните следующую строку кода:

```
char bigString[1000];      /* Никогда не знаешь, сколько памяти выделить... */
```

Раз уж мы занялись битами, нельзя оставлять это дело без внимания. Следовало сделать все корректно: узнать, сколько именно байт требуется, и выделить соответствующее количество памяти.

Разве не так?

Ведь в противном случае некий сообразительный хакер прочтет мой код и заметит, что я выделяю только 1000 байт и *надеюсь*, что этого хватит, и придумает какой-нибудь хитрый способ, чтобы с помощью `strcat` ввести 1100 байт в мои 1000 байт памяти и переписать кадр стека, изменив адрес

возврата. Тогда при возврате из функции выполнится код, написанный хакером. Именно это имеется в виду, когда говорят, что некая программа имеет уязвимость в виде *переполнения буфера*. Это было главной причиной взломов и червей в те достопамятные времена, когда Microsoft Outlook еще не сделал хакинг доступным даже подросткам.

Хорошо, значит, все эти программисты просто неучи. Им следовало посчитать, сколько памяти надо выделить.

Но язык C *не* предоставляет средств, с помощью которых это легко сделать. Вернемся к моему примеру с «The Beatles»:

```
char bigString[1000];        /* Никогда не знаешь, сколько памяти выделить... */
char *p = bigString;
bigString[0] = '\0';
p = mystrcat(p,"John, ");
p = mystrcat(p,"Paul, ");
p = mystrcat(p,"George, ");
p = mystrcat(p,"Joel ");
```

Сколько же памяти *надо* выделить? Попробуем сделать это правильным путем.

```
char* bigString;
int i = 0;
i = strlen("John, ")
   + strlen("Paul, ")
   + strlen("George, ")
   + strlen("Joel ");
bigString = (char*) malloc (i + 1);
/* не забыть место для завершающего нуля! */
...
```

Мои глаза стекленеют. Наверное, вы уже подумываете, не заняться ли чем-нибудь другим. Я вас не виню за это, но, поверьте мне, здесь начинаются действительно любопытные вещи.

Мы должны один раз просмотреть все строки, только чтобы выяснить их размер, а затем снова просмотреть их при конкатенации. Во всяком случае, для Pascal-строк операция strlen, вычисляющая длину строки, выполняется быстро. Может быть, нам удастся написать такую версию strcat, которая будет сама перераспределять память.

И тут мы открываем *еще один* ящик Пандоры, из которого вылетают функции выделения памяти. Вы знаете, как работает malloc? В основе malloc

лежит длинный связанный список свободных блоков памяти, называемый *цепочкой свободной памяти* (*free chain*). Функция malloc при вызове просматривает связанный список в поисках блока памяти, который достаточно велик для запроса. Затем она разрезает этот блок надвое – на блок указанного в запросе размера и блок с оставшимися байтами – и предоставляет вам тот блок, который вы просили, а оставшийся (если он есть) помещает обратно в связанный список. При вызове функции free освобождаемый блок помещается в цепочку свободной памяти. В итоге цепочка свободной памяти разрубается на мелкие кусочки, и когда потребуется выделить сразу много памяти, то участков такого размера не окажется в наличии. Тогда malloc берет тайм-аут и начинает рыться в цепочке свободной памяти, разбираясь в ситуации и сливая мелкие соседние свободные блоки в более крупные. На это уходит уйма времени. В результате всей этой суеты malloc всегда работает не слишком быстро (она обязательно просматривает цепочку свободной памяти), а иногда совершенно непредсказуемо начинает действовать очень медленно, когда производит уборку. (Это, между прочим, в точности соответствует системам со «сборкой мусора», поэтому сетования на то, что сборка мусора снижает производительность системы, не вполне справедливы, поскольку типичные реализации malloc приводили к такого же рода снижению производительности, хотя и более умеренному.)[1]

Сообразительные программисты минимизируют возможные отрицательные эффекты malloc, выделяя блоки памяти, размер которых является степенью двойки. Например, 4 байта, 8 байт, 16 байт, 18446744073709551616 байт и т. д. По причинам, интуитивно ясным всякому, кто имел дело с конструктором Lego, это минимизирует случайную фрагментацию в цепочке свободной памяти. Хотя может показаться, что это бесполезная трата памяти, но нетрудно убедиться, что напрасно тра-

[1] Здесь автор несколько сгущает краски, описывая простейшую реализацию. Проблема эта хорошо изучена (например, Д. Кнутом), и для нее найдены приемлемые решения: а) функция malloc может придерживаться не только стратегии «наилучший фрагмент», но и многих других: «первый достаточный», «самый большой» и т. д., которые существенно уменьшают сегментацию памяти; б) освобождение free может сразу объединять соседние свободные фрагменты, для чего фрагменты памяти организовываются не в односвязный список, а например, в двусвязный; в) динамическая сборка мусора должна отслеживать текущее количество актуальных ссылок на фрагмент, что много сложнее простого признака «занят – свободен». – *Примеч. науч. ред.*

тится не более 50% свободного пространства. В итоге программа тратит не более чем вдвое больше памяти, чем ей нужно, что не так уж страшно.

Допустим, что вы написали умную функцию strcat, которая автоматически выделяет для результата новый буфер. Должна ли она всегда выделять ровно тот объем, который требуется? Мой учитель и наставник Стэн Айзенштат (Stan Eisenstat)[1] предлагает при вызове realloc удваивать размер ранее выделенной памяти. Это означает, что вызывать realloc придется не более чем $lg(n)$[2] раз, что обеспечивает приемлемую производительность даже для очень больших строк, и при этом «пропадает» не более 50% памяти.

Ну да ладно. Когда имеешь дело с байтами, то все получается, как в поговорке «чем дальше в лес, тем больше дров». Хорошо все-таки, что мы не обязаны больше писать на C! У нас есть такие замечательные языки, как Perl, Java, VB и XSLT, благодаря которым не надо думать ни о каких байтах: они как-то сами с этим справляются. Но временами водопроводное хозяйство вылезает посреди жилой комнаты, и приходится думать над тем, какой выбрать класс – String или StringBuilder, или о чем-нибудь примерно таком же, потому что компилятор не настолько сообразителен, чтобы понять все наши замыслы, и пытается помочь нам *избежать* ненароком прокравшихся алгоритмов маляра Шлемиля.

В основе этой главы лежит статья, которую мне пришлось написать, т. к. в блоге я однажды сказал, что эффективная реализация оператора SQL SELECT author FROM books невозможна, если данные хранятся в формате XML.[3] Поскольку люди не поняли, о чем я тогда говорил, а мы как раз говорим об эффективности расходования памяти и ресурсов CPU, это утверждение может быть осмыслено лучше.

Каким образом реляционная база данных реализует SELECT author FROM books? В реляционной базе данных каждая строка таблицы (например, таблицы books) имеет одинаковый размер в байтах, и у каждого поля есть постоянное смещение от начала строки. Например, если все записи таблицы books имеют длину 100 байт, а поле author имеет смещение 23, то авторы хранятся в байтах, начиная с 23, 123, 223, 323 и т. д. Какой же код позволит перемещаться к следующей записи в результатах этого запроса? В принципе он имеет такой вид:

---

1 См. *www.cs.yale.ed/people/faculty/eisenstat.html*.

2 Логарифм по основанию 2 (в русскоязычной литературе в такой нотации принято записывать логарифм по основанию 10). – *Примеч. науч. ред.*

3 См. *www.joelonsoftware.com/articles/fog0000000296.html*.

```
pointer += 100;
```

Единственная команда CPU. Ну о-о-очень быстро.

Теперь взглянем на таблицу books в XML.

```
<?xml blah blah>
<books>
  <book>
    <title>UI Design for Programmers</title>
    <author>Joel Spolsky</author>
  </book>
  <book>
    <title>The Chop Suey Club</title>
    <author>Bruce Weber</author>
  </book>
</books>
```

Вопрос на сообразительность. Какой код позволит переместиться к следующей записи?

Да-да...

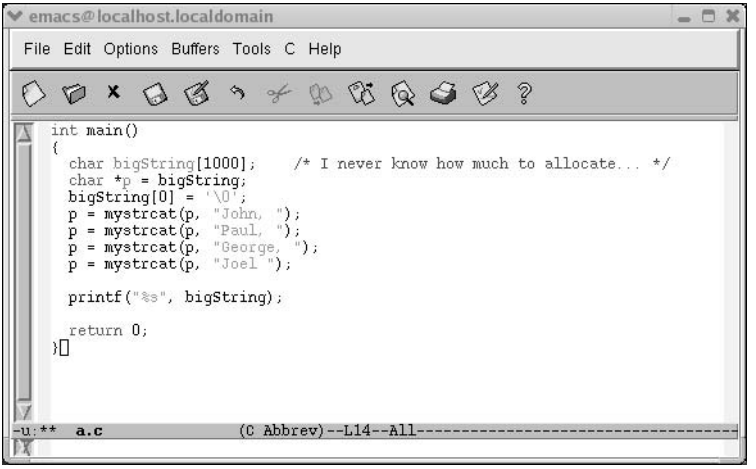
Хороший программист скажет здесь, что нужно выполнить анализ XML и создать в памяти дерево, с которым можно работать достаточно быстро. Объем вычислений, который при этом придется выполнить CPU, чтобы осуществить выборку `SELECT author FROM books`, приведет вас в полное отчаяние. Как известно любому разработчику компиляторов, лексический и синтаксический анализ составляют самую медленную часть компиляции. Достаточно отметить, что для лексического и синтаксического анализа и построения в памяти абстрактного синтаксического дерева требуются многочисленные операции со строками, которые, как мы выяснили, медленны, и многочисленные операции распределения памяти, которые тоже (как выяснили мы же) медленны. При этом предполагается, что памяти *достаточно* для одновременного хранения всей конструкции. В реляционных базах данных скорость перемещения от одной записи к другой постоянна и по сути определяется *одной командой CPU*. Так фактически и задумывалось. А благодаря отображению файлов в память достаточно загрузить с диска только те страницы, с которыми вы реально собираетесь работать. В XML же, если заранее провести синтаксический анализ, скорость перемещения от записи к записи будет фиксированной, но зато предварительная работа окажется огромной, а без предварительного анализа скорость перемещения к следующей записи будет зависеть от длины предшествующей записи, измеряясь сотнями команд ЦПУ.

Я делаю из этого вывод, что нельзя основываться на XML, если требуется высокая производительность и объем данных велик. Если данных немного или от программы не требуется высокое быстродействие, то XML может оказаться замечательным форматом. А если вы действительно хотите воспользоваться преимуществами того и другого, то вам нужно придумать способ хранения метаданных наряду с XML, что-то типа счетчика длины в Pascal-строках, который будет подсказывать, где что находится в файле, чтобы не нужно было для поиска проводить синтаксический анализ. Но тогда, конечно, нельзя будет изменять файл с помощью текстового редактора, поскольку метаданные придут в негодность, так что в действительности это уже будет не XML.

Если любезный читатель проследовал вместе со мной до этого места, то я надеюсь, что он узнал нечто новое для себя или переосмыслил известное. Полагаю также, что размышления над такими скучными темами из начального курса информатики, как фактическая реализация strcat и malloc, позволят вам по-новому взглянуть на принятие новых стратегических или архитектурных решений, когда вы столкнетесь с такими технологиями, как XML. В качестве домашнего задания поразмышляйте над тем, почему микросхемы Transmeta всегда будут казаться медленными. Или почему первоначальная спецификация таблиц в HTML была столь скверной, что большие таблицы на веб-страницах трудно было открывать тем, кто подключался к Интернету через модем. Или почему COM действует чертовски быстро, но только если не приходится пересекать границы процессов. Или почему разработчики NT поместили драйвер дисплея в пространство ядра, а не пользователя.

Все это вопросы, требующие размышлений на уровне байтов и оказывающие влияние на глобальные решения в отношении архитектуры и стратегии. Вот почему я думаю, что, преподавая информатику первокурсникам, надо начинать с самых азов – с работы на C и оттуда постепенно продвигаться вверх. Я испытываю совершенное отвращение к многочисленным программам обучения, в которых Java принят в качестве хорошего начального языка, потому что он «прост» и избавляет от мороки со строками и malloc, зато позволяет освоить все эти модные штучки ООП, благодаря которым большие программы становятся чрезвычайно модульными. Такое преподавание ведет нас к катастрофе. Мы окружены целыми поколениями университетских выпускников, придумывающих алгоритмы маляра Шлемиля на каждом шагу, ничуть этого не понимая, потому что они в принципе не понимают, что работа со строками на самом низком

уровне очень обременительна, несмотря на то, что из сценария Perl этого не видно. Если вы действительно хотите кого-то учить по-настоящему, надо начинать с самых основ. Как в «Малыше-каратисте». Полировать до блеска машину. И так три недели подряд. После этого свернуть кому-нибудь шею уже просто.



ГЛАВА ТРЕТЬЯ

# Тест Джоэла: 12 приемов написания лучшего кода

9 августа 2000 года, среда

Вы когда-нибудь слышали про SEMA?[1] Это довольно запутанная система оценки качества бригады разработчиков программного обеспечения. Нет, *погодите! Не нужно бросаться искать материалы о SEMA*! У вас лет шесть уйдет, чтобы *разобраться*, в чем там суть. Поэтому я создал свой собственный совершенно безответственный и грубый тест для оценки качества бригады разработчиков. Самое замечательное в нем то, что он отнимет у вас не более трех минут. Сэкономленное время вы можете потратить на то, чтобы окончить медицинский колледж.

### *Тест Джоэла*

1. Пользуетесь ли вы системой управления версиями исходного кода?
2. Можете ли вы выполнить сборку продукта за один шаг?
3. Выполняете ли вы ежедневную компиляцию?
4. Ведете ли вы базу данных ошибок в программе?
5. Исправляете ли вы ошибки, прежде чем писать новый код?
6. Есть ли у вас актуальный график работы?
7. Есть ли у вас спецификации?
8. Создали ли вы спокойные условия для работы программистов?
9. Стараетесь ли вы использовать для работы лучшие из существующих инструментов?

[1] См. *www.sei.cmu.edu/sema/welcome.html*.

10. Привлекаете ли вы к работе тестеров?
11. Предлагаете ли вы соискателям рабочих мест написать во время собеседования код?
12. Проводите ли вы проверку «юзабилити» на случайных людях?

Что привлекает в тесте Джоэла, так это возможность быстро ответить «да» или «нет» на каждый вопрос. Не надо вычислять количество строк кода, выдаваемых за день, или среднее количество багов на каждую модификацию программы. Добавьте себе одно очко за каждый ответ «да». К сожалению, тест Джоэла *явно не подходит* для проверки надежности программного обеспечения, написанного для работы атомной электростанции.

В идеале вы должны набрать 12 очков. 11 очков – терпимое количество, а 10 или менее указывают на серьезные проблемы. Фактически большинство софтверных организаций работает, имея лишь два или три очка, и они нуждаются в *серьезных* улучшениях, потому что такие компании, как Microsoft, постоянно держат 12 баллов.

Конечно, успех или поражение определяются не только этими факторами. В частности, если у вас прекрасная команда, которая работает над никому не нужным продуктом, то он так и останется никому не нужным. И наоборот, можно представить себе команду «гангстеров», не соблюдающих ни одного из перечисленных правил, и все же умудряющихся произвести поразительную программу, переворачивающую мир. Но, при прочих равных условиях, если эти 12 пунктов выполняются, значит, у вас есть дисциплинированная команда, способная стабильно выдавать готовый продукт.

## *1. Пользуетесь ли вы системой управления версиями исходного кода?*

У меня есть опыт работы как с коммерческими пакетами управления версиями, так и с CVS,[1] которая бесплатна, и должен сказать, что CVS – *отличный* продукт. Но если нет никакой системы управления версиями исходного кода, то наладить совместную работу программистов очень трудно. Программисты не смогут узнавать, что сделано другими. При обнаружении ошибки трудно вернуться к работающей версии. Еще одно замечательное свойство систем управления версиями заключается в том, что

[1] См. *www.cvshome.org/*.

сам исходный текст загружается на жесткий диск каждого программиста – я еще не слышал, чтобы в каком-нибудь проекте, где была задействована система контроля версий, потеряли большой объем кода.

## *2. Можете ли вы выполнить сборку продукта за один шаг?*

Здесь я имею в виду следующее: сколько шагов потребуется, чтобы собрать готовый к поставке продукт из последнего варианта исходного кода? В хороших организациях есть один сценарий, который достаточно запустить, чтобы сделать всю работу: загрузить весь исходный код, перекомпилировать каждую его строку, собрать все необходимые EXE-файлы для всех необходимых версий, языков и комбинаций #ifdef, создать пакет для инсталляции и подготовить окончательный носитель, будь то CD-ROM, веб-сайт для загрузки или нечто иное.

Если для этой процедуры требуется несколько шагов, то она чревата ошибками. Чем ближе срок выпуска продукта, тем желательнее иметь возможность очень быстро исправить «последнюю» ошибку, создать окончательные EXE-файлы и т. д. Если надо сделать 20 шагов, чтобы скомпилировать код, построить дистрибутив и т. д., вы просто сойдете с ума и наверняка наделаете глупых ошибок.

По этой самой причине последняя фирма, в которой я работал, перешла с Wise на InstallShield: мы *потребовали*, чтобы процедура инсталляции могла автоматически запускаться из сценария ночью с помощью планировщика NT, а Wise не могла запускаться ночью из планировщика, поэтому мы от нее отказались. (В компании Wise Solutions уверяют, что последняя версия их программы поддерживает ночные сборки.)

## *3. Выполняете ли вы ежедневную компиляцию?*

Программист иногда случайно добавляет что-то, что нарушает сборку. Например, новый файл исходного кода, и на его машине все прекрасно компилируется, но отправить этот файл в хранилище исходного кода он забывает. Он выключает свою машину и идет домой, счастливый и обо всем забывший. Но никто другой работать не может, и всем остальным тоже приходится идти по домам, но не такими счастливыми.

Представьте, что вы безуспешно пытаетесь скомпилировать продукт. Ситуация настолько ужасна и случается так часто, что компилировать имеет

смысл ежедневно, тогда неудача не пройдет незамеченной. В больших бригадах можно обеспечить своевременное исправление нарушений процедуры сборки, если проводить ее ежедневно, скажем, во время обеда. Перед обедом каждый программист загружает в систему как можно больше кода. Когда они возвращаются, компиляция уже завершена. Если она прошла успешно, все прекрасно. Все загружают себе последнюю версию исходного кода и продолжают работу. Если компиляция не прошла, ищут ошибки, но все могут продолжать работать с предыдущей, целой версией исходного кода.

У нас в бригаде разработчиков Excel было правило, что тот, по чьей вине произошла авария при сборке, в качестве «наказания» делался ответственным за все последующие сборки, пока кто-нибудь другой не сорвет очередную компиляцию. Это было хорошим стимулом для заботы о целости сборки и средством ротации, позволяющим каждому освоить процедуру сборки.

Подробнее про ежедневные сборки рассказано в моей статье «Ежедневная сборка – ваш друг».[1]

## *4. Ведете ли вы базу данных по программным ошибкам?*

Можете говорить что угодно по этому поводу. Но если вы разрабатываете ПО, то, пусть даже вы единственный его разработчик, без базы данных, содержащей все известные ошибки в коде, вы создадите продукт низкого качества. Многие программисты считают, что могут держать список ошибок в голове. Чепуха. Я не в силах запомнить более одной-двух ошибок одновременно, а на следующее утро или в спешке перед поставкой продукта они забываются. Совершенно необходимо вести формальный учет ошибок.

Базы данных по ошибкам могут быть довольно сложными или совсем простыми. Минимальная имеющая практическую пользу база данных должна содержать следующие сведения о каждой ошибке:

- последовательность действий, воспроизводящая ошибку;
- требуемое поведение;
- наблюдаемое (ошибочное) поведение;
- лицо, ответственное за исправление;
- отметка о том, исправлена ошибка или нет.

[1] См. главу 10.

Если от слежения за ошибками вас удерживает сложность соответствующего ПО, сделайте простую таблицу с пятью колонками для перечисленных важнейших полей и *начните ее заполнять*.

Подробнее о контроле над ошибками см. статью «Painless Bug Tracking» (Учет ошибок без всяких хлопот).[1]

## *5. Исправляете ли вы ошибки, прежде чем писать новый код?*

Самая первая версия Microsoft Word для Windows считалась «смертельным» проектом. Ему не было конца. Сроки нарушались постоянно. Вся бригада работала не покладая рук, но проект все не завершался, и напряжение было невероятным. Когда, наконец, с опозданием в годы вышел готовый продукт, Microsoft отправила всех разработчиков в отпуск в Канкун и занялась серьезной переоценкой ценностей.

В результате выяснилось, что менеджеры проекта настолько упорно старались выполнять расписание, что программисты торопились и писали крайне скверный код, поскольку фаза исправления ошибок отсутствовала в формальном графике. Не делалось никаких попыток вести учет ошибок. Совсем наоборот. Рассказывают, что один программист, который должен был написать функцию, вычисляющую высоту строки, просто написал `return 12;` и дождался появления отчета об ошибке, в котором говорилось, что его функция не всегда корректно работает. График работы был просто перечнем функций, которые еще не превратились в баги. В заключительном документе это было названо *методологией бесконечных ошибок*.

Для исправления ситуации Microsoft повсеместно распространила так называемую *методологию нулевых ошибок*. У многих программистов компании это вызвало смех, поскольку руководство, казалось, собиралось избавляться от ошибок в программах путем административных мер. На самом деле эта методология означала, что в любой момент времени устранение ошибок имеет более высокий приоритет, чем написание нового кода. И вот почему.

В целом, чем дольше откладывается исправление ошибки, тем дороже (по времени и по деньгам) обходится ее исправление.

Например, исправление опечатки или синтаксической ошибки, обнаруживаемой компилятором, происходит тривиальным образом.

---

[1] См. *www.joelonsoftware.com/articles/fog0000000029.html*.

Если вы нашли в своем коде ошибку при первой же попытке выполнить его, то можете сразу ее исправить, потому что весь код еще свеж у вас в памяти.

Если ошибка найдена в коде, написанном несколько дней назад, потребуется некоторое время, чтобы проследить ее возникновение, но, перечитывая написанный код, вы все вспомните, и исправление ошибки не отнимет много времени.

Если же ошибка найдена в коде, написанном несколько *месяцев* назад, то вы скорее всего изрядно его подзабыли, и исправить ошибку будет намного труднее. А может быть, придется исправлять код, написанный *кем-то другим*, кто сейчас отдыхает на Арубе, и тогда начнется что-то вроде научной работы, медленной, методичной и дотошной, и никто не сможет сказать, сколько времени займет поиск.

А если ошибка обнаружится в коде *уже распространяемого* продукта, то ее исправление обойдется неимоверно дорого.

Это одна из причин, по которым ошибки надо исправлять сразу: уходит меньше времени. Другая причина связана с тем, что легче *предсказать*, сколько времени потребуется для написания нового кода, чем для исправления имеющейся ошибки. Например, если я попрошу вас рассчитать, сколько времени потребуется, чтобы написать код для сортировки списка, вы дадите довольно точную оценку. Но если я предложу вам сказать, сколько времени уйдет на исправление ошибки, из-за которой ваш код не работает при установленном на машине Internet Explorer 5.5, то вы не ответите даже *приблизительно*, потому что не знаете (по определению), чем *вызвана* ошибка. Может быть, понадобится три дня, а может быть – две минуты.

Из этого следует, что если в вашем график работ много ошибок, подлежащих исправлению, то рассчитывать на его соблюдение нельзя. Но если все *обнаруженные* ошибки исправлены и остается лишь написать новый код, тогда график работ уже намного более точен.

Другая замечательная особенность немедленного исправления ошибок состоит в том то, что вы обретаете возможность значительно быстрее реагировать на действия конкурентов. Некоторые программисты считают, что это способ поддержания *готовности продукта к поставке*. Тогда вы сможете мгновенно ответить на вызов конкурента, предложившего новейшую, убийственную функцию, которая должна очаровать всех ваших покупателей, – вы сами реализуете такую же функцию и тут же выставите свой продукт, не задерживаясь на исправлении массы накопившихся ошибок.

## *6. Есть ли у вас актуальный график работ?*

Итак, мы добрались до графика работ. Если предполагается, что ваше ПО будет приносить какую-то прибыль, то срок готовности кода, конечно же, вызывает большой интерес. Известно, что программисты очень не любят составлять графики. «Когда напишу, тогда и будет готово!» – кричат они на управленцев.

К сожалению, такой ответ нельзя считать приемлемым. Соображения бизнеса диктуют необходимость принятия массы решений, связанных с планированием, задолго до поставки готового кода – необходимо подготовить демонстрационные программы, выставки, рекламу и т. д. А это невозможно без графика работ, поддерживаемого в актуальном состоянии.

График работ хорош еще и тем, что заставляет выбрать функции, которые подлежат реализации, а потом заставляет *пожертвовать* наименее важными из них и тем самым не дает нам впасть в «ползучий фичеризм».[1]

Вести график работы не так уж трудно. Прочтите главу 9, где описывается, как без хлопот составлять замечательные планы работ.

## *7. Есть ли у вас спецификации?*

Составление спецификаций похоже на чистку зубов с помощью флосса: никто не спорит, что это полезно, но никто этого и не делает.

Мне не совсем понятно, почему так происходит; возможно, дело в том, что большинство программистов очень не любят писать документацию. Если бригада состоит исключительно из программистов, перед которыми поставлена какая-то задача, они бросаются писать код, а не документацию. Они предпочитают углубиться в программирование, чем разрабатывать спецификацию.

Когда ошибка выявляется на стадии проектирования, ее легко исправить, изменив несколько строчек текста. А если код уже написан, то цена исправлений резко возрастает как в эмоциональном выражении (никто не любит выбрасывать код, который написал), так и единицах времени, поэтому никто не стремится на самом деле исправлять обнаруженные ошибки. Если программный продукт создается без спецификации, то он как правило плохо спроектирован и выпускается не вовремя. Надо полагать, что именно с такой бедой столкнулись в Netscape, когда первые четыре

[1] См. определение на *www.netmeg.net/jargon/terms/c/creeping_featuritis.html.*

версии пришли в полный беспорядок, и руководство бездумно решило выкинуть этот код и начать все сначала.[1] Ту же ошибку они повторили с Mozilla, создав неуправляемого монстра, для выпуска альфа-версии которого потребовалось *несколько лет*.

Согласно излюбленной мною теории эта проблема решается приучением программистов к писательскому труду путем отправки их на интенсивные курсы литературного мастерства.[2] Другим выходом может быть привлечение толковых администраторов, способных писать спецификации. Во всяком случае следует неукоснительно соблюдать правило, гласящее, что «код нельзя писать без спецификации».

О составлении спецификаций можно прочесть в главах с 5 по 8.

## *8. Создали ли вы спокойные условия для работы программистов?*

Написана масса книг и статей, в которых говорится, что производительность работников умственного труда, которым обеспечены достаточное пространство, тишина и уединенность, значительно возрастает. В классическом труде по управлению разработкой программ «Peopleware» подробно описаны достигаемые этим выгоды.[3]

Вот как действует этот механизм. Все мы знаем, что люди умственного труда лучше всего работают, когда «входят в ритм», когда им удается полностью сосредоточиться на работе и отключиться от окружающей обстановки. Они теряют всякое представление о времени и благодаря абсолютной концентрации создают замечательные вещи. В таком состоянии работа и бывает творческой. Писатели, программисты, ученые и даже баскетболисты подтвердят вам, что испытывают его.

Беда в том, что войти в это состояние не так просто. Если делать замеры времени, то окажется, что разогнаться до максимальной производительности труда можно примерно за 15 минут. Причем иногда, если вы устали или уже выполнили в этот день много творческой работы, вы вообще не

---

1 См. главу 24.

2 Например, в курсе «Daily Themes» в Йейльском университете (см. *www.yale.edu/engl450b/*) студенты должны ежедневно писать по одному эссе.

3 Tom DeMarco and Timothy Lister «Peopleware: Productive Projects and Teams», Second Edition, Dorset House Publishing, 1999 (Том Демарко и Тимоти Листер «Человеческий фактор: успешные проекты и команды», 2-е изд., СПб: Символ-Плюс, 2005).

сможете достичь этого состояния и будете бездельничать целый день, путешествуя по Интернету или играя в тетрис.

Другая проблема в том, что состояние вдохновения может легко оборваться. Шум, телефонные звонки, выход на обед, пять минут, чтобы попить кофе в Starbucks, вопросы коллег – *особенно* последнее! – все это выбивает из колеи. Чтобы ответить на вопрос, достаточно минуты, но вдохновение испарилось, и меньше чем через полчаса вы не восстановите максимальную производительность, да и вообще сделаете в этот день намного меньше. Если вы работаете в шумном загоне вроде тех, которые так любят устраивать в сумасшедших «доткомах», где люди из отдела маркетинга не отходят от телефонов, а программисты сидят с ними рядом, то продуктивность вашего умственного труда всегда будет низкой, потому что вас постоянно будут прерывать, и вы не сможете сосредоточиться.

Хуже всего программистам. Их продуктивность зависит от способности одновременно держать в кратковременной памяти массу мелких деталей. Любое постороннее вмешательство – и все эти детали улетучатся. Снова принимаясь за работу, вы не можете сразу вспомнить эти мелочи (типа имен локальных переменных или того места, до которого вы добрались, реализуя некий алгоритм поиска), и приходится все восстанавливать, что значительно замедляет работу.

Вот простейший пример. Из опыта известно, что если прервать работу программиста даже на одну минуту, то он не сможет продуктивно работать целых 15. Поместим двух программистов, Джеффа и Мэтта, в открытые соседние стойла обычной фермы для откорма телят (см. «Принцип Гилберта»). Мэтт забыл, как называется Unicode-версия функции `strcpy`. Он может поискать название в справочнике, на что уйдет 30 секунд, или спросить у Джеффа, на что уйдет 15 секунд. Поскольку Джефф сидит рядом, Мэтт обращается к нему. Джефф отвлекается и теряет 15 минут производительного труда (чтобы сберечь Мэтту 15 секунд).

А теперь переселим их в отдельные помещения со стенами и дверями. Забыв название этой функции, Мэтт может поискать его в справочнике, что по-прежнему займет 30 секунд, а может спросить у Джеффа, для чего потребуется уже 45 секунд, поскольку придется подняться с места (непростая задача, учитывая физическую подготовку среднестатистического программиста!) Итак, он посмотрит в справочнике. В итоге Мэтт потеряет 30 секунд, но зато Джефф сбережет 15 минут. Вот так!

## *9. Стараетесь ли вы использовать для работы лучшие инструменты?*

Создание кода из компилируемого языка – одна из немногих задач, которые все еще нельзя мгновенно выполнить на обычном домашнем компьютере. Если процедура компиляции занимает больше нескольких секунд, то лучше обзавестись новым и мощным компьютером и сберечь свое время. Даже если компиляция длится 15 секунд, программисты начинают скучать и почитывать «The Onion»,[1] увлекаясь настолько, что теряют целые часы производительного труда.

Отлаживать код GUI на машине с одним монитором мучительно, если вообще возможно. Если вы пишете код GUI, два монитора значительно облегчат вам жизнь.

Большинству программистов в конечном итоге приходится возиться с растровыми изображениями для пиктограмм и панелей инструментов, и у большинства программистов нет приличного редактора растровой графики. Редактировать графику с помощью Microsoft Paint – это издевательство, но большинству программистов приходится это делать.

На моем последнем месте работы[2] системный администратор регулярно присылал мне автоматическое почтовое сообщение с жалобой на то, что я использовал – вдумайтесь только! – больше 220 мегабайт дисковой памяти сервера. Я заметил ему, что при нынешних ценах на диски стоимость этой памяти существенно меньше стоимости израсходованной мною *туалетной бумаги*. Даже десять минут, потраченных на чистку каталога, окажутся непроизводительным расходом моего времени.

*В первоклассных командах не принято мучить программистов*. Даже мелкие неприятности, вызываемые плохим инструментом, копятся и делают программистов сердитыми и мрачными. А сердитый программист – малопродуктивный программист.

Вдобавок ко всему, программистов очень легко подкупать, приобретая для них самые крутые и новые вещицы. Это гораздо более экономный способ заставить их трудиться, чем платить им более высокие, по сравнению с другими фирмами, зарплаты!

---

[1] См. *http://www.theonion.com*.

[2] См. главу 32.

## *10. Привлекаете ли вы к работе тестеров?*

Если у вас нет специально выделенных тестеров, хотя бы одного на каждых двух-трех программистов, то либо вы выпускаете дырявые продукты, либо попусту тратите деньги, платя 100 долларов в час программистам за работу, которую тестеры могут выполнять за 30 долларов в час. Нежелание раскошелиться на тестеров оказывается такой ложной экономией, что я поражаюсь, как многие не могут этого понять. В главе 22 об этом говорится подробнее.

## *11. Предлагаете ли вы соискателям написать код во время собеседования?*

Интересно, наймете ли вы на работу фокусника, не попросив его сначала показать какие-нибудь фокусы? Разумеется, нет. А станете ли заказывать свадебный ужин в ресторане, не попробовав предварительно их кухню? Сомневаюсь. (Если только это не ваша тетя Мардж, которая возненавидит вас на всю жизнь, если вы не позволите ей испечь ее «знаменитый» пирог с ливером.)

И тем не менее, постоянно слышишь, что программиста приняли на работу только потому, что понравилось его внушительное резюме, или интервьюеру было приятно с ним поболтать. Либо задают кандидату мелочные вопросы («какая разница между CreateDialog() и DialogBox()?»), на которые легко найти ответ в документации. Надо интересоваться не тем, помнит ли человек кучу разных мелочей из программирования, а тем, в состоянии ли он писать код. Еще хуже, когда задают «коварные» вопросы, на которые легко ответить, если знаешь ответ, а если не знаешь, то ни за что не догадаешься.

Советую вам *все это прекратить*. Делайте во время интервью все что хотите, но заставьте испытуемого *написать какой-нибудь код*. (Более подробные рекомендации есть в моем «Справочнике бойца по проведению собеседований», приведенном в главе 20.)

## *12. Проводится ли юзабилити-тестирование на случайных людях?*

Вы хватаете за рукав первого попавшегося вам в коридоре человека и требуете, чтобы он поработал с кодом, который вы только что напи-

сали. Проделайте это пять раз, и вы найдете 95% всех проблем с юзабилити в своем коде.

Хороший интерфейс пользователя не так сложно разработать, как может показаться, и он совершенно необходим, если вы хотите, чтобы ваш продукт нравился клиентам, и они покупали его. Прочтите мою книгу о проектировании интерфейса пользователя[1] – краткий учебник для программистов.

При разработке пользовательских интерфейсов очень важно иметь в виду, что достаточно показать свою программу очень немногим людям (фактически достаточно пяти или шести человек) и сразу выяснить, какие наиболее серьезные проблемы возникают у пользователей. Прочтите статью Якоба Нильсена с соответствующими разъяснениями.[2] Даже если у вас отсутствуют навыки проектирования интерфейса пользователя, заставьте себя выполнить «коридорное» тестирование юзабилити (которое ничего не будет вам стоить), и вы значительно улучшите интерфейс.

## *Четыре способа применения теста Джоэла*

1. Оцените по этому тесту свою фирму и сообщите мне результат, а я уже начну распространять сплетни.
2. Если вы руководитель группы программистов, воспользуйтесь тестом как контрольным списком задач и добейтесь, чтобы команда стала работать как можно лучше. Достигнув результата в 12 баллов, можете оставить своих программистов в покое и полностью сосредоточиться на том, чтобы никакие коммерсанты им не докучали.[3]
3. Если вы устраиваетесь на работу в сфере программирования, поинтересуйтесь у предполагаемого работодателя, как он оценивает свою организацию по этому тесту. Если слишком низко, постарайтесь, чтобы вам предоставили право провести необходимые изменения к лучшему. В противном случае вас ждут разочарование и низкая продуктивность.

---

[1] Джоэл Спольски (Joel Spolsky) «User Interface Design for Programmers» (Проектирование интерфейса пользователя для программистов), Apress, 2001. Часть книги лежит в свободном доступе на *http://www.joelonsoftware.com/uibook/chapters/fog0000000057.html*.

[2] Якоб Нильсен (Jakob Nielsen) «Why You Only Need to Test with 5 Users» (Почему для проверки достаточно 5 пользователей), *useit.com*, 19 марта 2000 г. См. *www.useit.com/alertbox/20000319.html*.

[3] См. *www.joelonsoftware.com/articles/fog0000000072.html*.

4. Если вы инвестор и стараетесь оценить качество группы программистов, или ваша программная фирма рассматривает возможность слияния с другой, то этот тест позволит вам быстро оценить ситуацию.

## *Постскриптум: 14 июня 2004 года*

С момента появления «теста Джоэла» в августе 2000 года я получил массу писем от разработчиков со всего света с сообщениями о том, какую оценку получили их организации. Результаты сильно разнятся, но в подавляющем большинстве они составляют 2 или 3 балла. Н-да... Многие разработчики также сообщают о том, что сыты по горло практикой беспорядочной «гангстерской» разработки программ и отказываются от вакансий в компаниях, которые получают патологически низкие баллы по тесту Джоэла. Мне также сообщили о своих больших успехах некоторые менеджеры, использовавшие тест Джоэла для постепенного совершенствования производственного процесса.

В то же время, похоже, многие организации разработчиков пошли гораздо дальше, чем предлагает тест Джоэла, и теперь мучаются из-за прогрессирующего бюрократического атеросклероза. Верный признак его – трата на проведение совещаний большего времени, чем на разработку кода. Вполне реально получить по тесту Джоэла максимальные 12 баллов и потом настолько все испортить политическими играми и непроизводительными расходами, что прекратится всякая работа. Скажу по секрету, что с начала 1990-х годов, когда я работал в Microsoft, все свидетельствует о том, что в этой компании создались условия для застоя из-за ее огромных размеров, внутренней политики и расцветшего бюрократизма. Хотите доказательств? Возьмите Tablet PC – такое может понравиться только менеджерам среднего звена Microsoft: похоже, что они сконструированы для того, чтобы руководители программ в Редмонде могли совещаться дни напролет и при этом более-менее справляться с ежедневно получаемой порцией электронной почты. И это явление характерно не только для Microsoft. Если вы заметите, что слишком много времени тратите на установку и конфигурирование гигантских систем программной методологии или неких пакетов под названием «Визуальное нечто, архитектор масштаба предприятия», или постоянно обучаете своих разработчиков то экстремальному программированию, то UML, пока у них голова не начнет идти кругом, будьте готовы к неприятностям, даже если вы неплохо выглядите по тесту Джоэла.

ГЛАВА ЧЕТВЕРТАЯ

# Что каждый разработчик ПО должен(!) знать о Unicode и таблицах кодировки

8 ОКТЯБРЯ 2003 ГОДА, СРЕДА

Вы никогда не задумывались о том, что это за таинственный тег `Content-Type`? Тот, который полагается помещать в HTML, но в правильном значении которого никогда нет уверенности?

А приходилось вам получать электронную почту от друзей в Болгарии с темой сообщения «???? ?????? ??? ????»?

Я был поражен количеством разработчиков программ, которые плохо ориентируются в загадочном мире наборов символов, кодировок, Unicode и тому подобных вещей. Пару лет назад бета-тестер FogBUGZ[1] поинтересовался, справится ли эта программа с почтой на японском языке. Почтой на японском языке? У меня не было ни малейшего понятия. Присмотревшись внимательнее к коммерческому элементу ActiveX, с помощью которого мы анализировали сообщения электронной почты MIME, мы обнаружили, что он совершенно неправильно работал с кодировками, и пришлось написать код, который отменял неправильное преобразование, сделанное элементом, и выполнял свое, правильное. Заглянув еще в одну коммерческую библиотеку, я обнаружил, что и в ней работа с кодами символов реализована совершенно криво. Я связался с разработчиком пакета, который сообщил мне что-то типа «мы ничего не можем тут поделать». Как и многие другие программисты, он просто надеялся, что «как-нибудь пронесет».

---

[1] Наш продукт для слежения за ошибками; см. *www.fogcreek.com/FogBUGZ*.

Ничего не выйдет. Обнаружив, что PHP – популярный инструмент для веб-разработок – практически игнорирует вопросы кодировки символов[1] и беспечно работает с 8-битными символами, из-за чего почти невозможно разрабатывать добротные многоязычные веб-приложения, я сказал себе: *все, хватит.*

Поэтому делаю следующее заявление. Если вы – программист и работаете в двадцать первом веке, не зная основ таблиц символов, кодировок и Unicode, и я об этом *узнаю*, то накажу вас: будете сидеть полгода в подводной лодке и чистить лук. Клянусь.

И еще одно:

ЭТО НЕ ТАК УЖ ТРУДНО.

В данной статье я познакомлю вас с тем, что безусловно должен знать *всякий действующий программист.* Вся эта дребедень типа «обычный текст = ASCII = 8-битные символы» не просто неверна, а в корне неверна, и если вы все еще программируете таким образом, то не сильно отличаетесь от того врача, который до сих пор не верит в существование микробов. Не пишите никакого кода, пока не прочтете эту главу до конца.

Прежде чем начать, должен предупредить, что, если вы относитесь к тем редким людям, которые знакомы с вопросами интернационализации, мое изложение покажется вам несколько упрощенным. Я действительно постараюсь представить минимум сведений, чтобы каждый мог разобраться в них и писать код, который, *наверное*, сможет работать с любым языком, а не только с подмножеством английского, состоящим из слов без ударений. И еще предупреждаю, что обработка символов составляет лишь малую часть того, что требуется для создания интернациональных программ, но, поскольку я не могу писать обо всем сразу, то сегодня это будут кодировки символов.

## *Исторический экскурс*

Разобраться с этим материалом проще всего, если двигаться в хронологическом порядке. Вы, наверное, подумали, что я сейчас стану рассказывать о каких-нибудь очень старых кодировках типа EBCDIC. Нет, не буду. EBCDIC к вашей жизни не имеет никакого отношения. Так далеко углубляться мы не будем.

---

[1] См. *ca3.php.net/manual/en/language.types.string.php.*

В те не столь давние времена, когда разрабатывалась UNIX, а K&P писали «The C Programming Language»[1], все было просто. EBCDIC уходил в прошлое. Единственными нужными символами были старые добрые английские буквы без надстрочных значков, и для них существовал код под названием ASCII, в котором любой символ представлялся числом от 32 до 127.[2] Пробел имел значение 32, буква A – значение 65 и т. д. Они удобно помещались в 7 битах. Большинство компьютеров в те времена были 8-разрядными, а в одном байте можно было не только хранить любые символы ASCII, но и воспользоваться целым лишним битом в каких-нибудь своих порочных целях: разработчики WordStar, например, включали этот бит, чтобы обозначить последнюю букву слова, из-за чего WordStar был обречен работать только с английским текстом. Символы, коды которых были меньше 32, назывались *непечатными* и применялись для ругани. Шучу. Они выступали в качестве управляющих символов (например, 7 вызывает в компьютере звуковой сигнал, а 12 выкидывает печатаемую страницу из принтера и заправляет в него новую).

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | NUL | SOH | STX | ETX | EOT | ENQ | ACK | BEL | BS | HT | LF | VT | FF | CR | SO | SI |
| 1 | DLE | DC1 | DC2 | DC3 | DC4 | NAK | SYN | ETB | CAN | EM | SUB | ESC | FS | GS | RS | US |
| 2 | space | ! | " | # | $ | % | & | ' | ( | ) | * | + | , | - | . | / |
| 3 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | : | ; | < | = | > | ? |
| 4 | @ | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 5 | P | Q | R | S | T | U | V | W | X | Y | Z | [ | \ | ] | ^ | _ |
| 6 | ` | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o |
| 7 | p | q | r | s | t | u | v | w | x | y | z | { | \| | } | ~ | DEL |

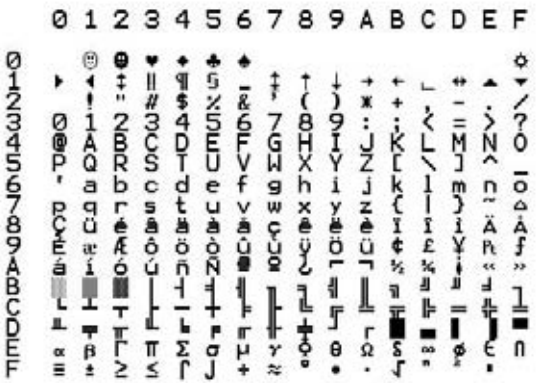
И все было хорошо при условии, что вашим родным языком был английский.

В байте, как известно, 8 бит, и многим пришла в голову мысль: «ха, мы можем использовать коды 128–255 для своих целей». Беда в том, что эта мысль пришла в голову *очень многим* одновременно, и у каждого было свое представление о том, что можно поместить в кодовые позиции от 128 до 255. В IBM-PC придумали нечто, получившее название кодировки OEM, в которой были некоторые символы с надстрочными знаками из европей-

[1] Brian Kernighan, Dennis Ritchie «The C Programming Language», Prentice Hall, 1978 (Брайан Керниган и Деннис Ритчи «Язык программирования C», СПб.: Невский Диалект, 2000).

[2] Подробнее об ASCII-символах см. *www.robelle.com/library/smugbook/ascii.html*.

ских языков и целый ряд символов для черчения – символов псевдографики, с помощью которых можно было рисовать на экране красивые окна и линии. Вы и сейчас можете их увидеть на компьютере с 8088-м процессором, если, конечно, найдете его. Как только начались продажи PC за пределами США, появилась масса кодировок OEM, в которых верхние 128 символов было разным. Например, на некоторых PC символ с кодом 130 отображался как й, а на компьютерах, продававшихся в Израиле, это была буква иврита Gimel (ג), поэтому rйsumйs, отправляемые американцами в Израиль, выглядели там как rגsumגs. Часто, например в русском языке, верхние 128 символов были разными даже для одного алфавита, поэтому надежного способа пересылки документов на русском языке не было.



В конечном итоге все это безобразие с OEM было узаконено в виде стандарта ANSI. Нижняя часть кодовой таблицы (до 128) практически совпадала с ASCII и не вызывала ни у кого возражений, но символов в верхней части (от 128) определялось множество, в зависимости от локализации. Эти различные системы были названы *кодовыми страницами*.[1] Например, в Израиле применялась кодовая страница DOS 862, а в Греции – кодовая страница 737. Нижние части таблиц совпадали, но все символы с кодами больше 128 были необычными. В переведенных на местные языки версиях MS-DOS были десятки таких кодовых страниц, и можно было работать с разными языками, от английского до исландского, и было даже несколько «мультиязычных» страниц, позволявших работать с эсперанто и галисийским *на одном и том же компьютере*! Но вот заставить, скажем, ото-

[1] Подробнее о кодовых страницах см. *www.i18nguy.com/unicode/codepages.html#msftdos*.

бражаться на одном и том же компьютере иврит и греческий было абсолютно невозможно без специальной программы, выводящей растровую графику, потому что ивриту и греческому требуются разные кодовые страницы с различной интерпретацией старших символов.

Тем временем в Азии приходилось сталкиваться с еще более удивительными вещами, если принять во внимание, что в азиатских алфавитах тысячи букв, которые никак не поместятся в 8 бит. Обычно для них применялась путаная система под названием DBCS, «двухбайтовый набор символов», в котором *одни* буквы занимали 1 байт, а другие – 2 байта. Перемещаться по строке в прямом направлении было легко, но в обратном – почти невозможно. Программистам рекомендовалось не пользоваться для перемещения операторами `s++` или `s--`, а вызывать особые функции, такие как `AnsiNext` и `AnsiPrev` в Windows, которые умели справляться с этой путаницей.

И тем не менее большинство людей делало вид, что байт – это символ, а символ – это 8 бит, и если вам не приходилось перемещать строки с одного компьютера на другой или говорить на нескольких языках одновременно, то это обычно сходило с рук. Естественно, с появлением Интернета перемещение строк с одной машины на другую стало обычным делом, и все пошло кувырком. К счастью, изобрели Unicode.

## *Unicode*

Unicode был смелой попыткой создать единый набор символов, включающий все имеющиеся на свете системы письма и даже некоторые вымышленные, такие как клингон. Некоторые пребывают в заблуждении, считая, что Unicode является просто 16-разрядным кодом, в котором, следовательно, может быть 65 536 различных символов. На самом деле это неверно. Это самый распространенный миф о Unicode, поэтому не расстраивайтесь, если вы тоже так думали.

В действительности, в Unicode иной способ представления символов, и если не разобраться, в чем его суть, то все остальное останется непонятным.

До сей поры мы предполагали, что буква отображается в какую-то группу битов, которые можно хранить на диске или в памяти:

```
A -> 0100 0001
```

В Unicode буква отображается в так называемую *кодовую позицию* – понятие чисто теоретическое. Механизм представления кодовой позиции в памяти или на диске – это уже другая история.

В Unicode буква A представляет собой платонический идеал. Она просто парит в небесах:

**A**

Платоническая **A** отличается от **B** и от **a**, но совпадает с A, *A* и A. Никто не спорит, что A в шрифте Times New Roman – это тот же символ, что A в шрифте Helvetica, но *не тот же*, что строчная буква «a», однако в некоторых языках *само понятие* буквы вызывает споры. Буква Я в немецком языке – это действительно буква или это особый способ записи ss? Если начертание буквы изменяется в конце слова, получаем ли мы другую букву? В иврите да, в арабском нет. Светлые головы в консорциуме Unicode выясняли эти вопросы лет десять, ведя при этом горячие политические споры, так что тут беспокоиться не о чем. Они уже все определили.

Консорциум Unicode присвоил каждой идеальной букве каждого алфавита магическое число, которое пишется, например, так: U+0645. Это магическое число называется *кодовой позицией*. U+ означает «Unicode», а цифры в этой записи шестнадцатеричные. U+0639 – это арабская буква Ain. Английской A соответствует код U+0041. Все эти числа можно узнать с помощью утилиты charmap в Windows 2000/XP или на сайте Unicode.[1]

Ограничений на количество символов, определяемых в Unicode, нет, и граница 65 536 фактически уже пройдена, так что не всякую букву Unicode можно затолкать в 2 байта – так это и было мифом.

Итак, пусть у нас есть строка:

```
Hello
```

В Unicode ей соответствуют следующие пять кодовых позиций:

```
U+0048 U+0065 U+006C U+006C U+006F
```

Всего лишь группа кодовых позиций. Просто числа. Мы еще ничего не сказали о том, как хранить их в памяти или помещать в почтовое сообщение.

## *Кодировки*

Вот где мы поговорим о *кодировках*. Самой первой идеей кодировки для Unicode, приведшей к появлению мифа о двух байтах, была такая: «Давайте запишем каждое число в два байта». Поэтому «Hello» превращается в:

[1] См. *www.unicode.org*.

00 48 00 65 00 6C 00 6C 00 6F

Правильно? Не спешите! А разве нельзя написать так:

48 00 65 00 6C 00 6C 00 6F 00 ?

Формально да, думаю, что можно, и первые разработчики приложений хотели иметь возможность записывать кодовые позиции Unicode в том порядке байтов (сначала старший или сначала младший), который был эффективнее на их конкретном CPU, и вот была ночь, и было утро, и уже появилось *два* способа записи Unicode. Поэтому пришлось придти к необычному соглашению: записывать в начало каждой строки Unicode байты FE FF, называемые «маркером порядка байтов Unicode», а если вы поменяете местами старший и младший байты, то получится FF FE, и тот, кто будет читать эту строку, будет знать, что и все остальные байты тоже нужно переставить местами.[1] Ну и ну. Реальные строки Unicode не всегда начинаются с маркера порядка байтов.

Какое-то время все шло хорошо, но потом программисты стали жаловаться. «Только посмотрите на эти бесчисленные нули!» – говорили они, потому что были американцами и читали английский текст, в котором редко встречались кодовые позиции дальше U+00FF. Кроме того, они были либеральными хиппи из Калифорнии и стремились к *бережливости* (*глумлюсь*). Будь они техасцами, их бы не беспокоил бесцельный удвоенный расход байтов. Но эти зануды из Калифорнии не могли видеть, как строки пожирают *вдвое больше* памяти, чем нужно, а кроме того, наплодилась масса документов в кодировках ANSI и DBCS (и кто все это будет конвертировать?). Поэтому большинство решило не обращать внимания на Unicode еще несколько лет, за которые положение стало еще хуже.

Так появилась блестящая идея UTF-8.[2] Система UTF-8 позволяла хранить строки кодовых позиций Unicode, этих волшебных чисел U+, в 8-разрядных байтах.[3] В UTF-8 каждая кодовая позиция от 0 до 127 записывается *в один байт*. Только кодовые позиции с номерами 128 и выше записываются 2, 3 или даже 6 байтами.

---

1 Подробнее о маркерах порядка битов см. *msdn.microsoft.com/library/default.asp?url=/library/en-us/intl/unicode_42jv.asp*.

2 См. *www.cl.cam.ac.uk/~mgk25/ucs/utf-8-history.txt*.

3 Подробнее о UTF-8 см. *www.utf-8.com/*.

| Hex Min | Hex Max | Последовательность байтов в двоичном виде |
|---|---|---|
| 00000000 | 0000007F | 0vvvvvvv |
| 00000080 | 000007FF | 110vvvvv 10vvvvvv |
| 00000800 | 0000FFFF | 1110vvvv 10vvvvvv 10vvvvvv |
| 00010000 | 001FFFFF | 11110vvv 10vvvvvv 10vvvvvv 10vvvvvv |
| 00200000 | 03FFFFFF | 111110vv 10vvvvvv 10vvvvvv 10vvvvvv 10vvvvvv |

При этом достигается удачный побочный эффект: английский текст выглядит в *UTF-8 точно так же, как в ASCII*, поэтому американцы даже ничего не заметят. Только всем остальным придется повертеться. А именно, строка Hello была у нас U+0048 U+0065 U+006C U+006C U+006F и будет записана как 48 65 6C 6C 6F, а именно так (обратите внимание!) она хранилась в ASCII, в ANSI и всех OEM-кодировках, какие только есть на свете. Теперь, если вы отважитесь использовать буквы со значками, или из греческого алфавита, или письма клингон, вам придется использовать для записи одной кодовой позиции несколько байтов, но американцам это безразлично. (У UTF-8 есть и такое милое свойство: невежественный старый код обработки строк, в котором нулевой байт играл роль терминатора, не станет усекать строки.)

Итак, я уже рассказал о *трех* способах кодировки Unicode. Традиционные способы записи в два байта называются UCS-2 (поскольку байтов 2) или UTF-16 (поскольку битов 16), и при этом надо еще выяснять, в каком порядке в этой UCS-2 записываются старший и младший байты. И есть еще популярный новый стандарт UTF-8, обладающий удачной возможностью прилично работать, если по счастливому стечению обстоятельств вам попались английский текст и одна из тех тупоголовых программ, которые не имеют никакого понятия о существовании других кодировок, кроме ASCII.[1]

На самом деле есть куча других кодировок Unicode. Например, нечто под названием UTF-7, очень похожее на UTF-8, но гарантирующее, что старший бит всегда будет нулем, поэтому если вам придется передавать строки Unicode по системе электронной почты какого-нибудь драконовского полицейского государства, где считается, что 7 бит *вполне достаточно*, то ваши строки смогут пройти невредимыми. Есть еще UCS-4, в которой каждая кодовая позиция хранится в 4 байтах, и удобным свойством которой является одинаковое количество байт для любой кодовой позиции, но так отчаянно транжирить память не позволят себе даже техасцы.

[1] См. *wwwzvon.org/tmRFC/RFC2279/Output/chapter2.html*.

Итак, вы научились думать в терминах идеальных платонических букв, представляемых кодовыми позициями Unicode, и можете кодировать эти кодовые позиции Unicode с помощью любой прежней системы кодировки! Например, можете закодировать строку Unicode для Hello (U+0048 U+0065 U+006C U+006C U+006F) в ASCII, или прежней греческой OEM-кодировке, или ANSI-кодировке для иврита, либо в любой другой из сотен имеющихся кодировок, *но с одной оговоркой*: может оказаться, что некоторые буквы не видны. Если для кодовой позиции Unicode, которую вы хотите изобразить, нет эквивалента в той кодировке, с помощью которой вы хотите это сделать, то обычно вместо него вы видите вопросительный знак (?). Или прямоугольник.

В сотнях известных традиционных кодировок лишь *некоторые* кодовые позиции хранятся правильно, а все остальные преобразуются в вопросительные знаки. В число популярных кодировок английского текста входят Windows-1252 (стандарт Windows 9x для западноевропейских языков) и ISO-8859-1, или Latin-1 (также удобная для всех западноевропейских языков).[1] Но попробуйте сохранить в этих кодировках русские или еврейские буквы, и вы получите кучу вопросительных знаков. UTF 7, 8, 16 и 32 обладают свойством правильно хранить *любые* кодовые позиции.

## *Самое главное о кодировках*

Если вы позабудете все, что я тут пытался объяснить, запомните, пожалуйста, один очень важный факт. *Строка не имеет никакого смысла, если вы не знаете, в какой кодировке она записана*. Не прячьте больше голову в песок и не делайте вид, что «обычный» текст – это ASCII.

**Нет такого понятия, как простой текст.**

Если у вас есть строка – в памяти, в файле или почтовом сообщении, то вы должны знать, в какой она кодировке, иначе вы не сможете правильно интерпретировать ее или показать пользователям.

Почти всегда у истоков дурацких проблем типа «мой веб-сайт выглядит как совершенный мусор» или «она не может читать мои письма, если в них

[1] Обзор набора символов ISO 8859-1 см. на *www.htmlhelp.com/reference/charset/*.

буквы с акцентами» стоит какой-нибудь наивный программист, который не понял простой вещи: если неизвестно, в какой кодировке – UTF-8, или ASCII, или ISO 8859-1 (Latin 1), или Windows 1252 (западноевропейская) – записана данная строка, то просто невозможно корректно показать ее или хотя бы найти ее конец. Известны сотни кодировок, и гадать, что находится в кодовых позициях выше 127, бесполезно.

Как сохранить информацию о том, в какой кодировке записана строка? Для этого существуют стандартные способы. В заголовок электронного письма надо поместить строку типа

```
Content-Type: text/plain; charset="UTF-8"
```

Для веб-страниц первоначально предполагалось, что веб-сервер должен возвращать аналогичный заголовок http Content-Type вместе со страницей – не в самом HTML, а в числе одного из служебных заголовков, посылаемых перед страницей HTML.

С этим связаны некоторые трудности. Допустим, что у вас есть большой сервер с множеством сайтов и сотнями страниц, написанных массой людей на разных языках и в тех кодировках, которые счел нужным сгенерировать тот экземпляр Microsoft FrontPage, который был у автора. Сам веб-сервер не сможет *узнать*, в какой кодировке записан каждый файл, и не сможет посылать заголовок Content-Type.

Удобно было бы поместить Content-Type файла HTML в сам файл HTML с помощью какого-нибудь специального тега. Конечно, поборники чистоты сойдут от этого с ума: как можно *прочесть* файл HTML, если неизвестно, в какой он кодировке?! К счастью, почти все общеупотребительные кодировки одинаково ведут себя с символами от 32 до 127, поэтому всегда можно добраться до этого места на странице HTML, не начав выводить странные символы:

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
```

Но тег meta действительно должен быть в самом начале раздела <head>, потому что веб-броузер, обнаружив этот тег, прекращает разбор страницы и возобновляет его после повторной интерпретации всей страницы в указанной вами кодировке.

Что делают веб-броузеры, если не находят Content-Type ни в заголовках http, ни в теге meta? Internet Explorer делает любопытную вещь: он пытает-

ся угадать язык и кодировку по частотам встречаемости байтов в типичных текстах с типичной кодировкой в различных языках. Такой алгоритм может оказаться успешным, поскольку в прежних 8-разрядных кодовых страницах буквы национальных алфавитов занимали разные диапазоны между 128 и 255 и поскольку для каждого естественного языка есть характерная гистограмма использования букв. Странно, но этот метод оказывается эффективным так часто, что наивные авторы веб-страниц, не имеющие представления о том, что у них должен быть заголовок Content-Type, смотрят на свою страницу в веб-броузере и думают, что все в порядке, пока в один прекрасный день не напишут нечто, не вполне соответствующее частотному распределению букв в их языке, и Internet Explorer не решит, что это корейский язык, и не отобразит страницу соответствующим образом. Из вышесказанного, как мне кажется, следует, что высказывание Джона Постела (Jon Postel) о необходимости «либерального отношения к тому, что принимаешь, и консервативного – к тому, что отправляешь» явно не может служить хорошим инженерным принципом.[1] И все-таки, что делает несчастный посетитель этого веб-сайта, написанного на болгарском языке, но показываемого как корейский (да и не корейский даже)? Он открывает пункт меню «Просмотр/Кодировка» и перебирает разные кодировки (для восточноевропейских языков их не меньше десятка), пока картинка не станет понятной. Если только он знает, что это можно сделать, потому что большинство этого не знает.

В последней версии CityDesk,[2] пакета для управления веб-сайтом, разработанного моей компанией Fog Creek Software, мы решили все внутренние операции осуществлять в UCS-2 (2-байтовой кодировке Unicode), которая принята в Visual Basic, COM и Windows NT/2000/XP в качестве основной кодировки строк. В коде C++ мы просто объявляем строки как wchar_t («wide char») вместо char и работаем с функциями wcs вместо функций str (например, wcscat и wcslen вместо strcat и strlen). Чтобы написать в коде C литеральную строку в кодировке UCS-2, достаточно предварить ее буквой L:

```
L"Hello"
```

Когда CityDesk публикует веб-страницу, он преобразует ее в кодировку UTF-8, которая уже много лет хорошо поддерживается веб-броузерами. Та-

[1] Приписывается Джону Постелу (Jon Postel). Из Information Sciences Institute, «RFC 791 – Internet Protocol», September 1981.

[2] См. *www.fogcreek.com/CityDesk*.

ким способом закодирована «Joel on Software» на 29 языках, и я не слышал, чтобы у кого-нибудь возникли трудности с их просмотром.[1]

Глава получилась довольно длинная, и мне едва ли удалось полностью осветить все аспекты кодировок символов и Unicode, но я надеюсь, что если вы дошли до этого места, то знаете достаточно, чтобы снова заняться программированием и лечить антибиотиками, а не пиявками и заклинаниями, каковому занятию я вас теперь и предоставляю.

[1] См. *www.joelonsoftware.com/navLinks/OtherLanguages.html*.

ГЛАВА ПЯТАЯ

# Безболезненное составление функциональных спецификаций

# Часть 1: стоит ли мучиться?

2 ОКТЯБРЯ 2000 ГОДА, ПОНЕДЕЛЬНИК

Когда впервые был обнародован тест Джоэла,[1] больше всего жалоб читателей было связано с составлением спецификаций. Как я уже говорил, спецификации похожи на флосс: все знают, что нужно их писать, но никто этого не делает.

Почему программисты не любят писать спецификации? Иногда они говорят, что так они экономят время. Как будто составление спецификаций – это роскошь, которую могут позволить себе только конструкторы космической техники в NASA или те, кто работает в крупных и известных страховых компаниях! Вздор. Прежде всего отсутствие спецификации – это *самый крупный и совершенно ненужный риск*, который вы берете на себя, принимаясь за программный проект. Это так же глупо, как отправиться через пустыню Мохаве, не вооружившись ничем, кроме обычной одежды, и надеясь, что как-нибудь обойдется. Программисты и разработчики, углубляющиеся в написание кода без спецификации, считают себя крутыми гангстерами, стреляющими от бедра. Ничего подобного. Они просто крайне непродуктивно работают. Они пишут плохой код, создают дрянные программы и угрожают своим проектам, подвергая их совершенно ненужному риску.

Я считаю, что в любом нетривиальном проекте (т. е. требующем больше недели кодирования или более одного программиста) отсутствие специ-

[1] См. главу 3.

фикации *всегда* приводит к потерям времени и снижению качества кода. Объясню почему.

Самая важная задача спецификации состоит в *проектировании программы*. Даже если вы работаете над кодом в одиночку и пишете спецификацию только для себя, процедура составления спецификации – детального описания функционирования программы – заставляет вас фактически проектировать программу.

Представим себе двух программистов в двух разных компаниях. Один из них, некто Пострел из софтверной фирмы «Скороспелые бананы», никогда не пишет спецификаций. «Что? Не надо нам никаких паршивых спецификаций!» А вот м-р Роджерс из фирмы «Софт без обмана» отказывается писать код, пока ему не выложат полную спецификацию. (Это всего лишь двое из массы моих воображаемых приятелей.)

Пострела и м-ра Роджерса объединяет одно: оба должны обеспечить обратную совместимость версии 2.0 своего программного продукта.

Пострел решает, что проще всего обеспечить обратную совместимость, написав конвертер для файлов, который преобразует их из версии 1.0 в версию 2.0. Он принимается за работу. Тюк-тюк-тюк. Клик-клик-клак. Диски крутятся. Пыль летит. Недели через две появляется сносный конвертер. Но клиенты Пострела недовольны. Код, написанный Пострелом, требует, чтобы вся фирма одновременно перешла на новую версию. Крупнейший клиент Пострела отказывается покупать новую программу. Он хочет, чтобы версия 2.0 могла работать с файлами версии 1.0, *не конвертируя* их. Пострел решает написать *обратный* конвертер и подключить его к функции «Сохранить». В результате возникают некоторые неприятности, потому что при использовании определенной функции версии 2.0 *создается впечатление*, что она работает, пока вы не соберетесь сохранить файл в формате 1.0. Лишь тогда вы узнаете, что функция, с которой вы работали полчаса назад, не действует в файле прежнего формата. В итоге потребовалось еще две недели на написание обратного конвертера, и работает он не очень здорово. Всего потрачено четыре недели.

Перейдем теперь к м-ру Роджерсу, представляющему «Софт без обмана», одному из тех занудных типов, которые отказываются писать код без спецификации. За 20 минут он придумывает такую же функцию обратной совместимости, какую сделал Пострел, и получает спецификацию, в которой сказано:

> Файл, созданный в прежней версии продукта, при открытии преобразуется в новый формат.

Эту спецификацию показывают клиенту, который говорит: «Э-э, нет, я не хочу, чтобы все сразу переходили на новую версию!» Поэтому м-р Роджерс думает еще некоторое время и исправляет спецификацию следующим образом:

> Файл, созданный в прежней версии продукта, при открытии преобразуется в новый формат в памяти. При сохранении файла пользователь получает возможность конвертировать его обратно.

Прошло еще 20 минут.

Босс м-ра Роджерса, помешанный на объектном программировании, видя эту спецификацию, решает, что тут что-то неправильно. Он предлагает другую архитектуру:

> Код надо переделать, создав в нем два интерфейса: V1 и V2. V1 будет содержать все функции версии 1, а в V2, наследующий V1, будут добавлены все новые функции. Тогда V1::Save справится с обратной совместимостью, а V2::Save сможет сохранить все добавленное. Если открыть файл V1 и попытаться воспользоваться функциональностью V2, программа сразу выдаст предупреждение, и пользователю придется конвертировать файл или отказаться от новой функциональности.

Прошло еще 20 минут.

М-р Роджерс ворчит. На этот рефакторинг у него уйдет три недели вместо двух, которые он планировал! Но зато элегантным образом будут решены все проблемы клиента, поэтому он соглашается и действует по этому плану.

Общее время, потраченное м-ром Роджерсом: три недели и один час. Время, потраченное Пострелом: четыре недели. И его код хуже.

Мораль этой истории в том, что вымышленным примером можно доказать все, что угодно. Ой... Я совсем не то хотел сказать. Мораль в том, что когда описываешь свой продукт обычным человеческим языком, то для того

чтобы представить себе различные возможности, исправить их и улучшить проект, нужны считанные минуты. Не слишком сложно убрать абзац в текстовом процессоре. Но если проектировать продукт на языке программирования, то на повторные проекты уходят *недели*. Хуже того, если программист уже потратил две недели на написание кода, то он держится за него, даже если код плохой. Никакими словами ни босс, ни клиенты не убедят Пострела выкинуть его прекрасный код конвертера, хотя он обеспечивает не самое лучшее решение. В итоге окончательный продукт оказывается неким компромиссом между первоначальным неудачным проектом и идеальным проектом. Получается «лучший продукт, который мы смогли создать, учитывая, что весь этот код уже написан, и мы просто не хотели его выкидывать». И это не совсем то же, что «лучший продукт, который мы смогли создать».

Это первая очень важная причина, по которой надо писать спецификации. Вторая важная причина состоит в том, что они *сокращают время, которое уходит на разговоры*. При составлении спецификации вы только *один раз* рассказываете, как должна работать программа. Все сотрудники могут просто прочесть спецификацию. В группе тестирования прочтут спецификацию и будут знать, как должна работать программа и что надо тестировать. В отделе маркетинга прочтут ее и напишут свои расплывчатые тексты, которые выложат на веб-сайты, рассказывая о еще не существующих продуктах. Менеджеры из отдела развития бизнеса поймут ее неправильно, и в их разгоряченном воображении продукт будет лечить облысение, бородавки и вообще все что угодно, но зато они привлекут инвесторов, ну и хорошо. Разработчики, прочитав ее, будут знать, какой код они должны написать. Клиенты прочитав ее, убедятся, что разработчики делают продукт, который стоит купить. Составители документации прочтут ее и напишут прекрасное руководство (его потеряют или выкинут, но это уже другая история).[1] Руководители прочтут ее и смогут сидеть на совещаниях со знающим видом. И так далее.

Если спецификации нет, то все описанные выше информационные процессы все равно идут, *потому что они не могут не идти*, но идут *под давлением обстоятельств*. Тестерам все равно придется возиться с про-

---

[1] См. главу о том, что никто ничего не читает, в книге Джоэла Спольски (Joel Spolsky) «User Interface Design for Programmers» (Проектирование интерфейса пользователя для программистов), Apress, 2001 или в Интернете по адресу *www.joelonsoftware.com/uibook/chapters/fog0000000062.html*.

граммой, но когда что-то покажется им странным, они придут к программистам и в *очередной раз* оторвут их от работы, задавая *новые* глупые вопросы о том, как должна работать та или иная функциональность. Это снижает продуктивность труда программистов,[1] а кроме того, они имеют привычку давать не «правильный ответ», а соответствующий тому, что написано у них в коде. В результате тестеры сравнивают программу с самой собой, а не с проектом, что было бы, простите, *несколько* полезнее.

Самое забавное (и в то же время печальное) это то, что происходит с несчастными составителями документации. У авторов документации часто отсутствует политический вес, позволяющий им прерывать работу программистов. Во многих компаниях, когда авторы документации начинают дергать программистов и спрашивать у них, как должна работать та или иная вещь, программисты идут к начальству и жалуются, что совершенно не могут работать из-за этих (опускаю брань) *писателей*, и *просят* их куда-нибудь *убрать*, а начальство, борясь за высокую производительность труда, запрещает авторам документации впредь красть время своих *дорогостоящих* программистов. Узнать такие компании очень просто, потому что у них в файлах подсказки или руководствах не больше информации, чем вы видите на экране. Там появилось сообщение:

> Включить поддержку LRF-1914?

Вы нажимаете кнопку «Help», появляется смешная (вы бы смеялись, если бы вам не было грустно) страница подсказки с таким примерно текстом:

> Позволяет выбрать между поддержкой LRF-1914 (по умолчанию) и отсутствием поддержки LRF-1914. Если вам нужна поддержка LRF-1914, щелкните «Yes» или нажмите «Y». Если вам не нужна поддержка LRF-1914, щелкните «No» или нажмите «N».

Да, большое спасибо. Вполне очевидно, что автор документации пытался скрыть, что *он не знает, что такое поддержка* LRF-1914. Он не мог спросить об этом программиста, потому что (a) ему было неловко, или (b) программист находился в Хайдарабаде, а автор в Лондоне, или (c) руко-

[1] См. *www.joelonsoftware.com/articles/fog0000000068.html*.

водство запретило ему прерывать программиста, или по какой-нибудь еще из многочисленных патологических причин, встречающихся в компаниях, но главная проблема в том, что *не было спецификации*.

Третья очень важная причина иметь спецификацию состоит в том, что без подробной спецификации нельзя составить график работ. Можно обойтись и без графика работ, если речь идет о вашей докторской диссертации, и вы собираетесь посвятить ей еще 14 лет, или если вы программист, работающий над очередной версией Duke Nukem, и *мы выпустим ее тогда, когда будем готовы*. Но практически любой вид бизнеса связан с необходимостью точно знать, сколько времени потребуется для решения задачи, потому что создание продукта стоит *денег*. Даже пару *джинсов* никто не купит, не поинтересовавшись их ценой, а как же ответственный руководитель может решить, создавать ли ему некий продукт, если неизвестно, сколько времени на это потребуется, а значит, во что это обойдется? Подробнее о графиках работ см. главу 9.

Очень распространенная ошибка – обсуждать, как сделать что-то, а затем *не принять никакого решения*. У Брайана Валентайна, ведущего разработчика Windows 2000, был знаменитый лозунг: «либо решение принимается в течение 10 минут, либо вопрос снимается с повестки».[1]

Очень часто в фирмах, создающих ПО, при обсуждении проекта никто не берет на себя смелость принять *решение*, обычно из политических соображений. Поэтому программисты работают только над тем, что не вызывает споров. В результате время идет, а трудные решения откладываются на конец. *Такие проекты – первые кандидаты на провал*. Если вы создали новую компанию с прицелом на какую-то новую технологию и поняли, что эта компания неспособна к принятию решений, можете сразу закрывать ее и возвращать деньги инвесторам, потому что конечный продукт никогда не будет создан.

Составление спецификации – хороший способ точно сформулировать все эти проектные решения, малыми и большими, которые вызывают такое раздражение, и о которых никто ничего не узнает, если спецификации нет. Спецификация помогает сформулировать даже мелкие решения. Допустим, вы создаете веб-сайт с регистрацией. Никто не спорит, что пользо-

---

[1] Пресс-релиз Microsoft «Valentine and His Team of 4,200 Complete the Largest Software Project in History» (Валентайн со своей командой в 4200 человек завершили крупнейший программный проект в истории) см. на *http://www.microsoft.com/presspass/features/2000/02-16brianv.mspx*.

вателю, забывшему пароль, надо отправить этот пароль по почте. Замечательно. Но этого мало, чтобы начать писать код. Чтобы написать код, надо знать, *какой текст* будет в этом почтовом сообщении. В большинстве компаний программистам не доверяют составлять тексты, адресованные пользователю (и, как правило, вполне обоснованно). Поэтому какой-нибудь маркетоид или рекламист, либо просто кто-то, кто знает язык, получает задачу составить точный текст письма. «Дорогой растяпа, вот пароль, который ты забыл. Постарайся впредь быть внимательнее». Если вы заставите себя написать *добротную и полную* спецификацию (а я расскажу об этом вскоре гораздо подробнее), то выявите все такие вопросы и либо решите их сразу, либо хотя бы обведете жирным красным карандашом.

Хорошо. Мы пока все там же. Написание спецификаций – это настоящая и подлинная добродетель. Подозреваю, что большинству это понятно, и мои речи хоть и забавляют вас, но не открывают ничего нового. Так почему же люди *не пишут* спецификаций? Не для того, чтобы сберечь время, ибо *этим его не сбережешь*, и большинству программистов, я думаю, это понятно. (В большинстве случаев на роль «спецификаций» нет других претендентов, кроме россыпи страниц, которые программист лепит в Блокноте *после того*, как напишет код, и *после того*, как объяснит уже трехсотому по счету человеку, что это за чертова функция.)

Я думаю, дело в том, что очень многие не любят писать. Созерцание пустого экрана повергает в сильную растерянность. Лично я преодолел свой страх перед писательством, посещая в колледже курс, на котором приходилось каждую неделю писать очерк на три-пять страниц. Писательство надо тренировать, как мускулы. Чем больше пишешь, тем лучше получается. Если вам надо написать спецификацию, и у вас ничего не выходит, начните вести дневник, создайте блог, пойдите на курсы литературного творчества или просто напишите хорошее письмо каждому из родственников или товарищей по колледжу, с которыми четыре года не общались. Любая деятельность, связанная с размещением слов на бумаге, улучшит вашу способность к написанию спецификаций. Если вы руководите разработкой программного продукта и те, кто должен у вас писать спецификации, не могут этого сделать, отправьте их на пару недель в горы на курсы литературного мастерства.

Если вам не доводилось работать в такой компании, где пишут функциональные спецификации, то вам, возможно, никогда и видеть их не приходилось. В следующей главе я приведу пример короткой спецификации, и мы обсудим, что должно присутствовать в хорошей спецификации.

ГЛАВА ШЕСТАЯ

# Безболезненное составление функциональных спецификаций

# Часть 2: что есть спецификация?

3 октября 2000 года, вторник

Я пишу о *функциональных*, а не о *технических* спецификациях. Их иногда путают. Не знаю, существует ли стандартная терминология, но вот что я подразумеваю под этими терминами:

> *Функциональная спецификация* описывает работу продукта исключительно с точки зрения пользователя. Она не касается того, как все это будет реализовано. В ней идет речь о функциях продукта. Она содержит описание экранов, меню, диалоговых окон и т. д.
>
> *Техническая спецификация* описывает внутреннее устройство программы. В ней идет речь о структурах данных, моделях реляционной базы данных, языках программирования, инструментарии, алгоритмах и т. д.

Разрабатывая всесторонний проект продукта, важнее всего определить условия работы пользователя – какие будут экраны, как они устроены, что они делают. Это позднее вы начнете беспокоиться о том, как это сделать. Нет никакого смысла спорить о том, какой язык программирования выбрать, пока вы не решили, что будет делать ваш продукт. В этой серии статей я говорю только о *функциональных спецификациях*.

Ниже приводится небольшой образец, который даст вам представление о том, как должна выглядеть хорошая спецификация.

## *Общие сведения*

WhatTimeIsIt.com – это служба, посредством которой можно узнать время в Интернете.

*Эта спецификация ни в коей мере не претендует на полноту.* Весь ее текст будет многократно пересмотрен, прежде чем примет окончательный вид. Графика и структура экранов приведены только для иллюстрации предлагаемой функциональности. Действительный внешний вид будет постепенно определяться усилиями графических дизайнеров и на основе мнений, сообщаемых пользователями.

Спецификация не касается алгоритмов, применяемых в механизме расчета времени и обсуждаемых в других документах. Она просто описывает, что видит пользователь, работая с сайтом *WhatTimeIsIt.com*.

## *Сценарии*

Проектируя продукт, полезно представить себе несколько реальных ситуаций, в которых обычные стандартные люди могли бы им воспользоваться. Мы рассмотрим два таких сценария.

### *Сценарий 1: Майк*

Майк – весьма занятой руководитель. Он президент крупной и известной фирмы, которая производит детские вещи, перерабатывая динамит, и распространяет их через общенациональные торговые сети. В обычный день у него бывает масса встреч с очень важными людьми. Иногда появляется сотрудник банка, угрожающий неприятностями за невыплату процентов по кредиту, полагавшуюся три месяца назад. Иногда приходит другой человек из другого банка, предлагающий ему открыть новую кредитную линию. Иногда его венчурные капиталисты (милые люди, давшие ему денег на открытие бизнеса) приходят к нему и сетуют, что он зарабатывает слишком много. «Костер! – требуют они. – Уолл-стрит любит смотреть на костры!»

Все эти посетители очень расстраиваются, если не могут найти Майка, назначившего им встречу. Все дело в том, что Майк не знает, который сейчас час. По совету своего секретаря Майк подписывается на WhatTimeIsIt.com. Теперь, если ему надо узнать время, он заходит

на сайт WhatTimeIsIt.com, вводит имя пользователя и пароль и узнает текущее время. За день он успевает побывать на этом сайте несколько раз – там он узнает, не пора ли пойти на ланч, не опаздывает ли он на очередную встречу и т. д. К концу дня, а фактически часов с трех пополудни, он заходит на сайт все чаще и чаще, чтобы не пропустить момент, когда можно будет отправиться домой. Начиная с 4:45 он практически непрерывно нажимает кнопку «Обновить».

*Сценарий 2: Синди*

Синди – старшеклассница. Она ходит в бесплатную среднюю школу и довольно сообразительна. Поэтому, возвращаясь домой в 2 часа дня, она тратит лишь около 7 минут (в среднем) на то, чтобы сделать домашнее задание по алгебре. Прочие учителя вообще ничего не задают на дом. Ее маленький брат сидит неотрывно перед единственным в доме телевизором и смотрит «телепузиков», поэтому оставшуюся часть дня (с 2:07 до 6:30, когда ее новая мама кормит всех обедом) она проводит в Интернете, болтая с друзьями в AOL. Она постоянно ищет интересные новые сайты. Случайно набрав слова «который теперь час?» в окне поисковой машины (по ошибке: она хотела спросить это у друзей в чате), она попала на WhatTimeIsIt.com и завела себе там учетную запись. Она вводит свое имя пользователя и «RyanPhillipe» в качестве пароля, указывает часовой пояс и – *вуаля* – узнает время.

## *Чего не будет в программе*

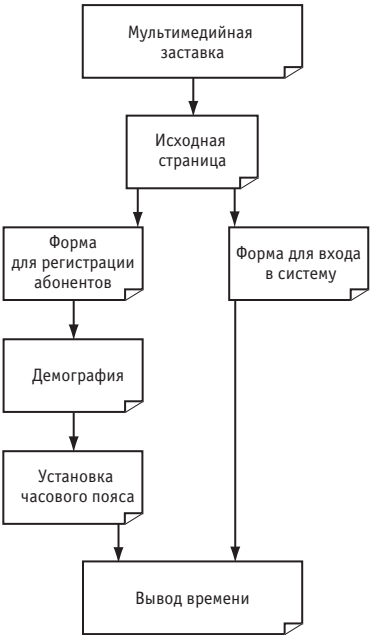
В данной версии *не* будут поддерживаться следующие функции:

- выбор нескольких часовых поясов для одного пользователя (предполагается, что каждый пользователь постоянно находится в одном и том же часовом поясе);
- изменение пароля;
- назначение встреч.

## *Блок-схема WhatTimeIsIt.com*

У нас еще будет время заняться изнурительными подробностями, а пока мы лишь взглянем на краткую блок-схему, чтобы представить

себе картину в целом. Блок-схема неполная, но правильно показывает, как мы собираемся использовать WhatTimeIsIt.com:




## *Спецификации для каждого экрана*

WhatTimeIsIt.com состоит из ряда совершенно разных экранов. Большинство из них будет соответствовать стандартному формату, а особенности стиля впоследствии разработает графический дизайнер. В данном документе больше внимания уделяется функциональности и интерактивности, а не окончательному внешнему виду.

Все экраны создаются в HTML. (Единственное исключение – экран заставки, созданный с помощью MacromediaShockwave).

Каждый экран WhatTimeIsIt.com имеет свое твердо установленное название, которое будет выделено в этом тексте жирным шрифтом, чтобы сразу было видно, что это название экрана, например **исходная страница**.

### *Экран заставки*

Неуместная и действующая на нервы анимация Shockwave, она играет дурацкую музыку и доводит всех до белого каления. **Экран заставки** мы закажем в дорогом *бутике* графической анимации, разместившемся на верхнем этаже здания в Сохо, куда люди приходят на работу с собаками, прикалывают себе к ушам английскими булавками разные мульки и *до ланча* успевают сбегать в Starbucks четыре раза.

После того как анимация покрутится секунд 10, в нижнем правом углу станет заметна ссылка «Пропустить этот экран». Всегда помещайте ее в самый угол, тогда ее никто не заметит и не щелкнет по ней. Она должна отстоять не меньше чем на 800 пикселов от левого края заставки и не меньше чем на 600 от верхнего.

Щелчок по кнопке «Пропустить этот экран» перемещает пользователя на **исходную страницу**. Когда кончается анимация, броузер переадресуется туда автоматически.

> **Нерешенный вопрос.** Если разрешит отдел маркетинга, мы будем помещать на компьютер пользователя cookie, когда он щелкнет по кнопке «Пропустить этот экран», и тогда впредь он не увидит заставку. Регулярных посетителей надо избавить от ее многократного просмотра. Я говорил об этом с Джимом из отдела маркетинга, и он собирается обсудить это на совещании комиссии по продажам, маркетингу и рекламе.

### *Исходная страница*

Отображается по завершении анимации Shockwave.

У исходной страницы три цели:

1. Дать людям представление об услуге и возможность решить, стоит ли им на нее подписываться.
2. Позволить уже подписавшимся абонентам зарегистрироваться на сайте.
3. Дать тем, кто решил подписаться, возможность создать для себя учетную запись.

**Исходная страница** имеет такой вид:


Щелчок по логотипу в левом верхнем углу этого и *вообще всех экранов* переводит пользователя обратно на **исходную страницу**.

> **Примечание.** Ввиду большого сходства между разными экранами надо задействовать какую-то систему *включений на стороне сервера*, и тогда, если изменится название службы или мы купим то доменное имя, которое хотим, можно будет поменять сразу все экраны, сделав исправления в одном месте. Предлагаю Vignette Story Server. Конечно, это излишество. Конечно, он стоит 200 000 долларов. Но с ним куда как проще иметь дело, чем с включениями на стороне сервера!

Ссылка с текстом «щелкните здесь, чтобы войти в систему» приводит на **форму входа в систему**. Ссылка с текстом «щелкните здесь, чтобы стать абонентом системы» приводит на **форму регистрации абонентов**. Остальные пять ссылок показывают страницы со статическим текстом, который предоставит администрация; они выходят за рамки данной спецификации и меняются редко.

### *Форма входа в систему*

С помощью **формы входа в систему** зарегистрированный абонент вводит данные своей учетной записи и получает возможность узнать текущее время. Она имеет следующий вид:


Правая часть экрана обеспечивает такие же возможности, какие описаны выше для **исходной страницы**.

Окно адреса электронной почты позволяет ввести до 60 символов, окно пароля – до 12. Вместо вводимых с клавиатуры символов отображаются звездочки (*), что помогает защитить пароль.

> **Примечание.** Это достигается с помощью тега `<INPUT TYPE=PASSWORD>`.

Когда пользователь щелкает по кнопке «Зарегистрироваться», сервер выполняет следующие проверки:

1. Если адрес электронной почты указан, но он недействителен, поскольку неверен его формат (нет знака @ или есть символы, запрещенные для почтовых адресов в RFC-822), сервер возвращает новую страницу, которая выглядит в точности, как **форма входа в систему**, но над окном для адреса выведено красным цветом сообщение об ошибке с текстом «Недопустимый адрес. Пожалуйста, проверьте его правильность». Этот текст выведен красным цветом, но текст «Пожалуйста, введите свой адрес электронной почты» по-прежнему выводится чер-

ным. Некорректный адрес, ранее указанный пользователем, остается в поле ввода.

2. Если адрес электронной почты введен, но такой адрес не принадлежит ни одному зарегистрированному абоненту, сервер возвращает новую страницу, которая выглядит в точности как **форма входа в систему**, но над окном для адреса выведено красным цветом сообщение об ошибке с текстом «Введенный адрес не принадлежит зарегистрированным абонентам. Пожалуйста, проверьте его правильность. Чтобы стать зарегистрированным абонентом, щелкните по ссылке в правой части экрана». Этот текст выведен красным, но текст «Пожалуйста, введите свой адрес электронной почты» по-прежнему выводится черным. Некорректный адрес, ранее введенный пользователем, уже присутствует в текстовом окне. (Вопрос к разработчикам. Можно ли здесь с помощью JavaScript сделать так, чтобы при щелчке пользователя по ссылке, ведущей на **форму регистрации абонентов**, почтовый адрес на ней заполнялся автоматически?)
3. Если введен адрес электронной почты, и он принадлежит одному из зарегистрированных абонентов, но не был введен пароль, посылаем по этому адресу почтовое сообщение, содержащее пароль. Тема сообщения «Ваше членство в WhatTimeIsIt.com». Сообщение имеет *обычный текстовый формат*. Точный текст бурно обсуждается советом директоров и будет сообщен ближе к моменту готовности продукта. (Разработчикам: у меня есть собственный злобный текст. Он заставит Чака подпрыгнуть на стуле.)
4. Если введен адрес электронной почты, и он принадлежит одному из зарегистрированных абонентов, и введен пароль, но пароль неправильный, сервер возвращает новую страницу, которая выглядит в точности, как **форма входа в систему**, но над окном для пароля выведено красным цветом сообщение об ошибке с текстом «Введенный пароль не верен. Пожалуйста, проверьте его правильность. Не забывайте правильно задавать *верхний или нижний регистр символов*». Если в пароле нет символов нижнего регистра, то дополнительно выво-

дим текст «Может быть, вы случайно нажали клавишу CAPS LOCK?» Если пароль неверен, то **форма входа в систему** возвращается с *пустым* окном для ввода пароля.

5. Если адрес электронной почты и пароль верные, сразу переходим на **страницу времени**.

**Нерешенный вопрос.** Надо принять решение по поводу JavaScript в пункте 2.

**Нерешенный вопрос.** Надо получить текст сообщения с паролем от руководителя компании.

Вот что я помещаю в каждую спецификацию.

**Отказ от ответственности.** Чистая страховка. Какая-нибудь фраза, сообщающая, что это неполная спецификация или что-нибудь в этом роде, может спасти от нашествия разъяренных людей на ваш офис. Со временем спецификация совершенствуется, и можно уже сообщить, что «я старался сделать эту спецификацию как можно более полной, но если я что-то упустил, пожалуйста, сообщите мне». Это напоминает о том, что у каждой спецификации должен быть:

**Автор.** Один автор. В некоторых фирмах считают, что спецификации должны составляться *коллективно*. Если вам когда-нибудь доводилось заниматься «коллективным творчеством», то вы знаете, что худшей пытки не бывает. Предоставим заниматься коллективным творчеством фирмам, консультирующим по менеджменту: в них трудятся армии молодых выпускников Гарварда, которым нужно демонстрировать кипучую деятельность, чтобы оправдывать свои баснословные оклады. Ваши спецификации должны быть написаны *одним человеком*, и он должен за них отвечать. Если создаваемый вами продукт велик, разбейте его на части, и пусть за спецификацию каждой части отвечает отдельный человек. В некоторых компаниях полагают, что эгоистично и против правил совместной работы оказывать честь одному человеку, помещая его имя на спецификации. *Чушь.* Человек должен брать на себя *ответственность* за предлагаемую им спецификацию. Если с ней что-то окажется не в порядке, то человек, несущий за нее ответственность, имя которого напечатано в спецификации, и должен ее исправлять.

**Сценарии.** При проектировании продукта надо иметь несколько реальных сценариев его использования. Иначе он может не найти никакого реального применения (как :CueCat).[1] Определите аудитории для своего продукта и представьте себе фиктивного, совершенно искусственного, но совершенно *стандартного* пользователя в каждой из них, который будет пользоваться продуктом совершенно стандартным образом. В главе 9 моей книги о проектировании интерфейсов пользователя[2] рассказывается о создании фиктивных пользователей и сценариев. Вот здесь они и должны быть. Чем живее и реалистичнее сценарии, тем полезнее проект вашего продукта для реальных или воображаемых пользователей, поэтому я и стараюсь вставлять побольше выдуманных подробностей.

**Антизадачи.** При групповой разработке продукта каждый участник проекта предлагает свои любимые реальные или мнимые функции, без которых, как ему кажется, не обойтись. Если воплощать их все, то потребуется немыслимое количество времени и денег. Отсеивание функций надо начинать сразу, и лучше всего в разделе «антизадачи» спецификации. Вы просто перечисляете в нем то, чего *делать не будете*. Это может быть функция, которую вы не станете реализовывать («никаких телепатических интерфейсов пользователя!» или нечто более общее («В этой версии мы не стремимся к быстродействию продукта. Пусть он получится медленным, но работающим. Если позволит время, то при разработке версии 2 мы оптимизируем медлительные процессы».) Эти антизадачи могут вызвать определенные дискуссии, но надо поскорее создать ясность.

**Общий обзор.** Это нечто вроде оглавления спецификации. Можно выполнить его в виде простой блок-схемы или пространного описания архитектуры. С ним должны ознакомиться все, чтобы получить общую картину, тогда детали станут более понятны.

**Детали, детали, детали.** Наконец мы добрались до деталей. Большинство читателей пропустят их в поисках конкретной подробности, интересующей именно их. Когда вы проектируете некий веб-сервис, удобно дать каждому экрану твердое название и написать разделы, подробно и дотошно их описывающие.

*Детали* – это самое важное в функциональной спецификации. Вы видели в приведенном примере, как я самым подробным образом описывал

---

1 См. *www.joelonsoftware.com/articles/fog0000000037.html*.

2 Spolsky, Joel (Джоэл Спольски) «User Interface Design for Programmers» (Apress, 2001). См. *www.joelonsoftware.com/uibook/ chapters/ fog0000000065.html*.

все возможные ошибки для страницы входа в систему. Что если почтовый адрес неправильный? Что если пароль неверный? Все эти варианты соответствуют реальному коду, который надо будет написать, но еще важнее, что они соответствуют *решениям*, которые кто-то должен будет принимать. Кто-то должен решить, какая политика действует в случае забытого пароля. Если вы этого не решите, то не сможете написать код. Это решение должно быть зафиксировано в спецификации.

**Нерешенные вопросы.** Ничего страшного, если в первой версии спецификации останутся открытые вопросы. Когда я пишу первый вариант, то всегда оставляю массу открытых вопросов, но помечаю их (с помощью особого стиля, чтобы потом можно было их найти) и, если есть такая возможность, описываю альтернативы. К тому времени, когда начнут работу программисты, все они должны быть решены. (Если вы считаете, что можно дать программистам начать работать над простыми вещами, а тем временем разбираться с открытыми вопросами, то это плохое решение. Слишком много *новых* проблем возникнет, когда программисты начнут писать код, чтобы иметь еще старые, которые были известны давно и могли быть решены. Кроме того, решение по любой нетривиальной проблеме может сильно повлиять на способ написания кода.)

**Заметки на полях.** Разрабатывая спецификацию, учитывайте различные аудитории, которым она предназначена: программистов, тестеров, специалистов по маркетингу, составителей документации и т. д. Во время этого процесса вам в голову могут приходить различные соображения, полезные для какой-нибудь одной из этих групп. Например, я помечаю информацию для программистов, которая обычно касается каких-то деталей технической реализации, заголовком «Примечание». Специалисты по маркетингу такие замечания пропускают. Программисты же с жадностью набрасываются. Мои спецификации обычно полны разных «замечаний по тестированию», «замечаний по маркетингу» или «замечаний по составлению документации».

**Спецификация должна оставаться живым документом.** Иногда в программистских фирмах встречается «каскадный» образ мыслей. Мы, дескать, сначала спроектируем программу, напишем спецификацию, напечатаем ее, а потом отдадим программистам и пойдем по домам. Могу сказать только одно: «Ха-ха-ха-ха-ха-ха!»

Именно из-за этого спецификации имеют дурную славу. Мне часто говорят: «спецификации бесполезны, потому что им никто не следует, они всегда устаревшие и никогда не соответствуют реальному продукту».

Извините. Это, может быть, *ваши* спецификации устаревшие и не соответствуют реальному продукту. Мои спецификации регулярно обновляются. Обновление происходит по мере разработки продукта и принятия новых решений. Спецификация всегда наилучшим образом отражает наше общее представление о том, как будет работать продукт. Спецификация перестает изменяться только тогда, когда написан код продукта (т. е. реализована функциональность, но остаются тестирование и отладка).

Чтобы не слишком осложнять людям жизнь, я не пересматриваю спецификации ежедневно. Обычно я держу где-то на сервере последнюю версию спецификации, чтобы разработчики могли с ней сверяться. Периодически я распечатываю спецификацию, помечая в ней места, подвергшиеся изменению, чтобы не приходилось читать все заново: можно поискать отметки о ревизии и узнать, что появилось нового.

ГЛАВА СЕДЬМАЯ

# Безболезненное составление функциональных спецификаций

# Часть 3: но... как?

4 ОКТЯБРЯ 2000 ГОДА, СРЕДА

Теперь, когда вы знаете, зачем нужна спецификация и что она должна содержать, поговорим о том, кто должен ее составлять.

## *Кто пишет спецификации?*

Расскажу небольшую историю из жизни Microsoft. В 1980-х там начался бурный рост, и все прочли «Мифический человеко-месяц», классическую книгу по теории менеджмента в разработке программного обеспечения.[1] (Если вы ее не читали, настоятельно рекомендую это сделать.) Главная идея книги была в том, что если подключать новых программистов к проекту, который не укладывается в график, то итоговая задержка только увеличится. Дело в том, что если в проекте участвует $n$ программистов, то количество каналов общения между ними равно $n(n-1)/2$ и растет, как $O(n^2)$.

Поэтому программисты Microsoft были обеспокоены противоречием между необходимостью писать все более крупные программы и господствовавшей тогда теорией, что добавление новых программистов только ухудшает положение.

Чарльз Симонайи, долгое время бывший главным архитектором Microsoft, предложил назначать *главных программистов*. Суть идеи была в том, что один старший программист отвечает за написание всего кода, но у него есть бригада подчиненных программистов-невольников. Вместо того

[1] Фредерик Брукс «Мифический человеко-месяц», СПб.: Символ-Плюс, 2000.

чтобы самому отлаживать каждую функцию, старший программист пишет прототип каждой функции, создавая ее эскиз, и отдает на доработку младшим программистам. (Разумеется, сам Симонайи должен был стать Главным Старшим программистом.) От термина «старший программист» (master programmer) веяло средневековьем, поэтому в Microsoft его заменили на «менеджера программы» (program manager).

Это нововведение было призвано решить проблему «Мифического человеко-месяца», ведь теперь не требовалось, чтобы все общались друг с другом: каждый младший программист общается только с менеджером программы, поэтому объем коммуникаций растет как $O(n)$, а не $O(n^2)$.

Что ж, Симонайи, вероятно, был знаком с венгерской нотацией,[1] но не читал книгу Тома Демарко и Тимоти Листера «Peopleware».[2] Никто не хочет писать код на положении раба. Система оказалась совершенно неработоспособной. В конечном счете Microsoft обнаружила, что все-таки можно подключать к проекту толковых программистов и при этом увеличивать продуктивность работы, хотя и с убывающей эффективностью (и хотя в «Мифическом человеко-месяце» утверждалось обратное). Когда я работал в команде Excel, она состояла из 50 программистов и была продуктивнее, чем была бы бригада из 25 человек, но не *вдвое* продуктивнее.

Идея проектирования по схеме «хозяин/невольник» была дискредитирована, но люди, которых называли «менеджерами программ», не исчезли из Microsoft. Должность менеджера программы по сути была реанимирована сообразительным человеком по имени Джейб Блюменталь. Отныне менеджер программы должен был заниматься *проектированием* и *спецификацией* продуктов.

Теперь менеджеры программ в Microsoft собирают технические требования, определяют, что должен делать код, и *пишут спецификации*. Обычно на каждого из них приходится пять программистов; эти программисты должны реализовывать в коде то, что менеджер программы реализовал в виде спецификации. Менеджер программы также координирует маркетинг, составление документации, тестирование, локализацию и другие

---

[1] Система специальных соглашений по конструированию имен переменных в программе, например iFixCore. Существует мнение, что венгерская нотация предложена Симонайи и впервые была широко использована именно разработчиками MicroSoft. – *Примеч. науч. ред.*

[2] Tom DeMarco, Timothy Lister «Peopleware: Productive Projects and Teams», Second Edition, Dorset House Publishing, 1999 (Том Демарко и Тимоти Листер «Человеческий фактор: успешные проекты и команды», 2-е изд., СПб.: Символ-Плюс, 2005).

противные мелочи, на которые не должны тратить свое время программисты. Наконец, предполагается, что менеджеры программ Microsoft должны видеть задачи компании в целом, а программисты могут сосредоточиться на том, чтобы безукоризненно писать свой код.

Значение менеджеров программ трудно переоценить. Если вы недовольны тем, что программисты больше озабочены технической элегантностью, а не вопросами маркетинга, значит, вам нужен менеджер программы. Если вы недовольны тем, что люди, которые умеют писать хороший код, не в состоянии писать хорошим литературным языком, вам нужен менеджер программы. Если вы недовольны тем, что продукт развивается в каком-то неясном направлении, вам нужен менеджер программы.

## *Как принимать на работу менеджера программы?*

В большинстве компаний нет даже представления о должности менеджера программы. Думаю, это совершенно неправильно. В мое время те группы в Microsoft, где был сильный менеджер программы, выпускали очень успешные продукты: сразу приходят в память Excel, Windows 95 и Access. Зато другими группами (например, MSN 1.0 и Windows NT 1.0) управляли разработчики, обычно игнорировавшие менеджеров программ (которые были не слишком хороши и, возможно, заслуживали того, чтобы их игнорировали), и их продукты были не очень удачными.

Старайтесь соблюдать следующие три «не»:

1. **Не выдвигать кодеров в менеджеры программ.** Те навыки, которые нужны хорошему менеджеру программы (умение писать ясным языком, дипломатические качества, знакомство с маркетингом, сочувствие к пользователю и проектирование хороших интерфейсов), очень редко сочетаются с навыками, необходимыми хорошему кодеру. Есть, конечно, люди, способные к обоим видам деятельности, но они встречаются редко. Поощрение хороших кодеров выдвижением их на *другие должности*, где требуется писать на человеческом языке, а не на C++, служит классической иллюстрацией принципа Питера: люди продвигаются по службе, пока не достигнут своего уровня некомпетентности.[1]
2. **Не выдвигать специалистов по маркетингу на должность менеджера программы.** Здесь нет ничего оскорбительного, но я думаю,

[1] Peter, Laurence with Hull, Raymond (Лоуренс Питер и Рэймонд Холл) «The Peter Principle» (Принцип Питера), Buccaneer Books, 1996.

что мои читатели согласятся: у хороших специалистов по маркетингу редко присутствует достаточное понимание технических проблем, необходимое для проектирования продуктов.

По существу менеджер программы – это отдельная специальность. Он должен быть хорошо подготовлен в технической области, но ему не обязательно быть хорошим кодером. Менеджеры программ изучают интерфейс пользователя, встречаются с клиентами и *пишут спецификации*. Они должны ладить с самыми разными людьми – от «слабоумных» клиентов до раздражающих программистов-нелюдимов, приходящих на работу в одеяниях из Star Trek, и расфуфыренных коммерсантов в костюмах за 2000 долларов. В некотором смысле менеджер программы соединяет группы разработчиков программ в одно целое. Он должен быть харизматической личностью.

3. **Не требовать, чтобы кодеры *отчитывались* перед менеджером программы.** Это неочевидная ошибка. В качестве менеджера программы в Microsoft я разрабатывал стратегию Visual Basic (VBA) для Excel и составил полную, подробную спецификацию, описывающую реализацию VBA в Excel. Объем моей спецификации приближался к 500 страницам. В момент самой интенсивной работы над Excel 5.0 я прикинул, что ежедневно 250 человек приходили на работу и по существу реализовывали ту громадную спецификацию, которую я написал. Я не мог знать всех этих людей; одних лишь *составителей документации* для этого проекта в бригаде Visual Basic было около десятка (и целая группа писала документацию со стороны Excel, а один человек был постоянно занят только контролем гиперссылок в файле подсказки). Удивительно, что при этом я был в самом низу административной иерархии. Серьезно, мне *никто* не подчинялся. Если я хотел, чтобы что-то было сделано, мне приходилось убеждать людей в том, что это нужно. Когда Бен Уолдмен, ведущий разработчик, не хотел делать то, что было написано в моей спецификации, он просто не делал этого. Если тестеры жаловались, что какую-то часть моей спецификации полностью протестировать нельзя, я вынужден был упрощать ее. *Если бы кто-нибудь из этих людей мне подчинялся, продукт получился бы хуже.* Иные из них считали неуместным спорить с начальником. В других случаях я упирался и *приказывал* сделать по-моему (из-за тщеславия или недальновидности). Обычно мне не оставалось ничего иного, кроме как согласиться. Такая форма принятия решений – лучший способ *получить хороший результат.*

ГЛАВА ВОСЬМАЯ

# Безболезненное составление функциональных спецификаций

# Часть 4: советы

15 октября 200 года, воскресенье

Итак, мы обсудили, зачем нужны спецификации, что они должны содержать и кто должен их писать. В четвертой и последней части этой серии я поделюсь некоторыми советами о том, как писать хорошие спецификации.

Чаще всего там, где действительно пишут спецификации, жалуются на то, что их никто не читает. Когда никто не читает спецификации, их авторы становятся несколько циничными. Как на старой карикатуре, где инженеры надстраивают свои кабинки в офисе стопками четырехдюймовых спецификаций. В типичной большой бюрократической организации все месяцами пишут нудные спецификации. Написав, их отправляют навсегда на полку, а проект делается с нуля и без какой-либо оглядки на спецификацию, потому что ее никто не читал, потому что *от нее мозги сохнут*. Собственно, процесс написания спецификации мог оказаться полезным, потому что мог хотя бы заставить всех задуматься над стоящими проблемами. Но составленная спецификация была отправлена на полку, ее никто не прочитал, и она никому не понравилась, поэтому все и думают, что это был мартышкин труд.

Кроме того, если никто не читал спецификацию, то по завершении работы над продуктом возникает много споров. Кто-нибудь (из руководства, клиентов или из отдела маркетинга) восклицает: «Погодите, но вы же обещали сделать пароварку для моллюсков! Где она?» А программисты отвечают: «Никак нет, если вы заглянете в главу 3 спецификации, подраздел 4,

параграф 2.3.0.1, то увидите, что там прямо сказано – никаких пароварок для моллюсков». Но клиент этим не удовлетворяется, поскольку он всегда прав, и разозленным программистам приходится дополнительно оснащать продукт пароваркой для моллюсков (что делает их еще более циничными в отношении спецификаций). Либо менеджер заявляет: «Послушайте, в тексте этого диалогового окна слишком много слов, а еще в верхней части каждого диалогового окна должна быть реклама». Программисты в отчаянии: «Но ведь вы сами *утвердили спецификацию*, в которой *точно* указаны текст и расположение элементов каждого диалогового окна!» Разумеется, менеджер не читал спецификацию, потому что, когда он попытался это сделать, у него чуть не произошло размягчение мозга, и вообще во вторник он играл в гольф и у него не было времени.

Итак. Спецификации вещь хорошая, но только тогда, когда их читают. Если вы пишете спецификацию, то должны *сделать так*, чтобы ваш труд был прочтен, а также постараться, чтобы скромные остатки чьих-нибудь мозгов не вытекли окончательно из ушей их обладателя.

Чтобы соблазнить людей чтением вашего труда, как правило достаточно написать его хорошим языком. Но будет нечестно, если я скажу «пишите лучше» и на этом покину вас. Вот пять простых правил, которые абсолютно *необходимо* соблюдать, чтобы ваши спецификации читали.

## *Правило 1: Пишите занимательно*

Да, это первое правило, которое должен соблюдать тот, кто хочет соблазнить людей чтением своей спецификации (читатели должны получить удовольствие). Не говорите мне, что у вас от рождения нет дара писать занимательно, я вам не поверю. У всех постоянно возникают смешные мысли, но внутренний цензор ворчит, что это «непрофессионально». Иногда необходимо нарушать правила.

Если вы видели тонны той ерунды, которую я написал для своего вебсайта, то должны были обратить внимание, что я постоянно, то здесь, то там, пытаюсь быть занимательным. Чуть выше я грубо пошутил над чужими мозгами и поиздевался над менеджерами, любящими играть в гольф. Не будучи большим весельчаком, я все же стараюсь шутить, ведь даже *попытки* писать забавно оживляют текст, как бывает смешон печальный клоун. В спецификации проще всего быть занимательным, приводя примеры. Рассказывая о том, как работает некая функция, не говорите:

> Чтобы создать новую таблицу «Служащие», пользователь нажимает клавиши Ctrl+N и начинает вводить их имена.

Лучше написать примерно так:

> Мисс Пигги, тыча в клавиатуру карандашом для подведения глаз, потому что ее маленькие пухлые пальчики слишком толсты, чтобы нажимать клавиши по отдельности, набирает Ctrl+N, чтобы создать таблицу «Бойфренды», и вводит в нее единственную запись «Kermit».

Если вы достаточно часто читали Дэйва Барри,[1] то могли обратить внимание, что один из простейших способов быть занимательным – это быть *конкретным* тогда, когда этого не требуется.[2] «Мопсы всех мастей» звучит забавнее, чем «собаки». «Мисс Пигги» звучит забавнее, чем «пользователь». Вместо «пользователей с особыми запросами» напишите о «фермерах-левшах, выращивающих авокадо». Не говорите «тех, кто не хочет убирать за своими собаками, следует наказывать»; лучше скажите, что «их следует помещать в такие глухие тюрьмы, где заключенным приходится покупать секс у пауков».

Между прочим, если вы считаете, что быть занимательным – это непрофессионально, то у вас, извините, просто нет чувства юмора. (И не спорьте. Те, у кого нет чувства юмора, всегда это отрицают. Вам меня не провести.) А если вы работаете в такой компании, где вас будут меньше уважать, если ваши спецификации станут живыми, забавными и приятными для чтения, то лучше поищите себе другую компанию,[3] потому что жизнь так *чертовски коротка*, что жаль проводить светлое время суток в таком суровом и несчастном месте.

## *Правило 2: Спецификация должна напоминать код, предназначенный для выполнения мозгами*

Программистам трудно даются хорошие спецификации, и дело тут, я думаю, вот в чем.

---

[1] Dave Barry – писатель-юморист, лауреат Пулитцеровской премии. – *Примеч. ред.*

[2] См. *www.miami.com/mld/miamiherald/living/columnists/dave_barry/*.

[3] См. *www.fogcreek.com/*.

Главной аудиторией написанного *кода* оказывается *компилятор*. Да, я знаю, что людям тоже приходится читать код, но обычно им очень трудно это делать.[1] Большинству программистов достаточно сложно довести код до такого состояния, когда компилятор может прочесть его и правильно интерпретировать; желание сделать код пригодным для чтения человеком оказывается роскошью. Код может быть таким:

```
void print_count( FILE* a, char  *  b, int c ){
   fprintf(a, "there are %d %s\n", c, b);}
main(){ int n; n =
10; print_count(stdout, "employees", n) /* код
намеренно сделан непонятным*/ }
```

или таким:

```
printf("there are 10 employees\n");
```

В обоих случаях будет напечатано одно и то же. Поэтому, как мне кажется, чаще всего встречаются программисты, которые пишут в следующем стиле:

> Допустим, что имеется функция AddressOf(x), определяемая как отображение пользователя x, в RFC-822-совместимый почтовый адрес этого пользователя, представляющий собой ANSI-строку. Предположим далее, что имеются пользователи A и B, причем A должен послать почтовое сообщение пользователю B. Для этого пользователь A инициирует новое сообщение с помощью любого (но не всех сразу) из определенных в соответствующих местах способов и вводит AddressOf(B) в текстовом окне To: .

Ведь можно было написать иначе:

> Мисс Пигги собирается на обед, поэтому она создает новое почтовое сообщение и вводит адрес Кермита в поле «To:».
>
> **Примечание:** Это должен быть стандартный адрес электронной почты (согласно RFC-822).

[1] См. главу 24.

*Теоретически* «смысл» в обоих случаях одинаков, но только первый пример невозможно понять без тщательного дешифрования, а второй пример понять легко. Программисты часто пытаются писать спецификации, похожие на непостижимые научные статьи. Программисты иногда думают, что «корректная» спецификация должна быть «формально» корректной, и уж тут только держись.

Ошибка состоит в том, что при написании спецификации помимо корректности необходима еще и *понятность*, что на программистском языке означает необходимость написать ее так, чтобы человеческий мозг смог ее «скомпилировать». Одно из существенных различий между компьютером и человеческим мозгом состоит в том, что компьютер может спокойно ждать, пока вы определите термины, которые собрались использовать. Но человек не поймет без предварительных объяснений, о чем вы говорите. Человек не собирается что-то *расшифровывать*, он хочет читать по порядку и понимать. Человеку надо сначала показать общую картину, а *затем* уже расписывать детали. Компьютерную программу вы начинаете сверху и двигаетесь вниз, полностью излагая детали. Компьютеру безразлично, имеют ли смысл имена ваших переменных. Человеческий мозг схватывает идею гораздо лучше, если может составить живую картинку по вашему рассказу, даже если это лишь часть рассказа, потому что наш мозг эволюционировал так, чтобы понимать рассказы.

Если опытному шахматному игроку показать позицию, возникшую на доске в середине игры, то он за пару секунд запомнит расположение всех фигур. Но если переставить несколько фигур так, чтобы возникла позиция, которая неосуществима в реальной игре (например, поставить какие-нибудь пешки на первую горизонталь или обоих черных слонов поставить на черные клетки), то запомнить такое расположение фигур на доске шахматисту будет несравнимо труднее. Компьютеры думают по-другому. Компьютерная программа, которая может запомнить шахматную доску, с одинаковой легкостью запомнит как возможные, так и невозможные позиции. Человеческому мозгу *не* свойственна работа по принципу прямого доступа; какие-то пути в нашем мозгу оказываются более прочными, и одни вещи проще понять, чем другие, потому что они встречаются чаще.

Поэтому, когда вы пишете спецификацию, постарайтесь представлять себе того, кому она адресована, и что должно быть сделано для него понятным на каждом шаге. С каждой новой фразой старайтесь выяснить, сможет ли читающий *понять ее* во всей глубине в контексте того, что вы уже ему сообщили. Если кто-либо из ваших предполагаемых читателей может

не знать, что такое RFC-822, то вы должны либо дать определение, либо хотя бы упрятать упоминание об RFC-822 в примечание, чтобы какой-нибудь административный работник не прекратил чтение спецификации, столкнувшись с обилием технического жаргона.

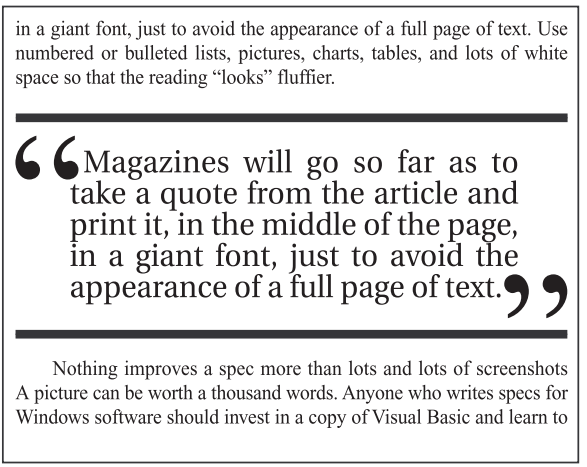
## *Правило 3: Пишите проще*

Не пишите неестественным и формальным языком из опасения, что простые предложения выглядят непрофессионально. Язык спецификации должен быть максимально простым.

Часто люди употребляют такие слова, как «утилизировать», потому что просто «использовать» кажется им непрофессиональным. (Опять это слово – «непрофессиональный». Будьте уверены, что когда кто-то говорит, что не следует чего-то делать, потому что это «непрофессионально», у него просто нет *настоящих* аргументов.) На самом деле, мне кажется, что часто люди, встретив текст, написанный понятным языком, чувствуют в этом какой-то подвох.

Разбивайте текст на короткие предложения. Если не удается написать понятное предложение, разбейте его на два или три более коротких.

Избегайте чрезмерно длинного текста – целых страниц, на которых нет ничего, кроме текста. Люди пугаются их и бросают читать. Часто ли вы видели популярный журнал или газету, в которых целая страница была бы заполнена текстом? В журналах иногда даже выхватывают цитату из статьи и печатают ее огромным шрифтом в центре страницы – лишь бы гла-


in a giant font, just to avoid the appearance of a full page of text. Use numbered or bulleted lists, pictures, charts, tables, and lots of white space so that the reading “looks” fluffier.

“Magazines will go so far as to take a quote from the article and print it, in the middle of the page, in a giant font, just to avoid the appearance of a full page of text.”

Nothing improves a spec more than lots and lots of screenshots A picture can be worth a thousand words. Anyone who writes specs for Windows software should invest in a copy of Visual Basic and learn to

зам не представала страница, заполненная исключительно текстом. Вставляйте нумерованные или маркированные списки, рисунки, диаграммы, таблицы и пустые промежутки, чтобы придать странице более легкомысленный вид.

Ничто так не полезно в спецификации, как обилие скриншотов. Одна картинка может быть ценнее тысячи слов. Если вы пишете спецификации программ для Windows, приобретите Visual Basic и научитесь работать с ним хотя бы в объеме, позволяющем делать макеты экранов. (На Mac воспользуйтесь REAL Basic; веб-страницы макетируйте с помощью FrontPage или Dreamweaver.) Затем сделайте скриншоты (Ctrl+PrtSc) и вставьте их в свою спецификацию.

## *Правило 4: Исправляйте и перечитывайте текст по нескольку раз*

Поначалу я готовился долго рассуждать по поводу этого правила, но оно слишком просто и очевидно. Перечитайте и отредактируйте свою спецификацию несколько раз. Если обнаружите предложение, которое не удается понять *с первого раза*, переделайте его.

Я сэкономил столько времени, отказавшись от разъяснения правила 4, что могу добавить еще одно правило.

## *Правило 5: Шаблоны бывают вредны*

Не поддавайтесь соблазну сделать стандартный шаблон для спецификаций. Вам может показаться, что хорошо, если «все спецификации выглядят одинаково». Сообщаю: не хорошо. Какой в этом смысл? А книги у вас в доме все выглядят одинаково? А вы бы хотели, чтобы они выглядели одинаково?

Плохо то, что при наличии шаблонов в них часто появляются разделы, которые, как вам кажется, необходимы для каждой функции. Пример: большой Билл заявляет, что отныне в каждый продукт Microsquish будет включена интернет-компонента. Поэтому в шаблоне спецификации теперь есть раздел с названием «Интернет-компонента». Теперь всякий, составляя спецификацию, как бы тривиальна она ни была, должен написать этот раздел «Интернет-компонента», даже если это просто спецификация для клавиатуры Microsquish. (А вы удивляетесь, почему на клавиатурах, как грибы, стали расти ненужные кнопки для покупок в Интернете).

По мере того как копятся разделы, растет размер шаблона. (Сошлюсь на крайне плохой шаблон спецификации, который можно видеть на *www.construx.com/survivalguide/desspec.htm*. Ну кому нужен в спецификации список литературы? Или глоссарий?) Такие большие шаблоны плохи тем, что отбивают желание писать спецификации, и это занятие начинает пугать.

Спецификация – это документ, который хотелось бы, чтобы люди прочли. И в этом он не отличается от рассказа в «Нью-Йоркере» или дипломной работы. Вы когда-нибудь слышали, чтобы профессор раздавал студентам шаблоны дипломных работ? Вы когда-нибудь читали два хороших рассказа, написанных по шаблону? Откажитесь от этой мысли.

ГЛАВА ДЕВЯТАЯ

# График работ без всяких хлопот

29 МАРТА 2000 ГОДА, СРЕДА

В октябре 1999 года северо-восток Соединенных Штатов пестрил рекламой «Acela» – нового поезда-экспресса между Бостоном и Вашингтоном. Широкая реклама на ТВ, билбордах и плакатах вызывала мысль, что на новую дорожную услугу Amtrak *какой-то* спрос должен появиться.

Возможно, так оно и произошло бы. Выяснить это Amtrak не удалось. Пуск Acela был отложен один раз, потом другой, рекламная кампания закончилась, а услуга Acela так и не начала действовать. Это напоминает мне слова одного маркетолога, которые он произнес, когда его продукт получил восторженные отзывы за месяц до появления в продаже: «Какой успех! Жаль только, что нельзя *купить* эту чертову штуку!»

Одуревшие от тестостерона компании-производители игр любят хвастаться на своих сайтах, что очередная игрушка начнет поставляться, «как только будет готова». График работы? Какой еще график?! Мы крутые кодеры! Большинству компаний такая роскошь недоступна. Возьмите Lotus. Когда они впервые изготовили 123 версии 3.0, для нее требовался 80286-й компьютер, который в те времена был не очень распространен. Тогда они отложили выпуск продукта на 16 месяцев, в течение которых пытались втиснуть его в 640K памяти, бывших пределом для 8086. Когда они это сделали, Microsoft получила фору в 16 месяцев в разработке Excel, а 8086, по иронии судьбы, отошел в прошлое!

В тот момент, когда я это пишу, выпуск веб-броузера Netscape 5.0 опаздывает почти на *два года*. Отчасти это вызвано их самоубийственно ошибочным решением выкинуть весь код и начать все заново: та же ошибка, кото-

рая отправила на свалку истории Ashton-Tate, Lotus и Apple MacOS. За этот период доля пользователей броузера Netscape сократилась с 80 до 20 процентов, и решать какие-либо проблемы конкурентной борьбы он не мог, потому что его главный программный продукт был разобран на тысячу кусочков, разбросанных по полу, и был ни на что не годен. Главным образом одно это неправильное решение стало той бомбой, с помощью которой Netscape покончила с собой. (Подробности можно узнать у Джеми Завинского, громко высказавшего свое возмущение;[1] к этой теме я вернусь в главе 24.)

Итак, *обязательно составляйте график работы*. Почти никто из программистов не любит этого делать. Мой опыт показывает, что большинство из них старается обойтись вообще без графика. Те немногие, кто составляют график, обычно делают это только под давлением начальства, неохотно, и никто *не верит* в графики на практике, исключая высшее руководство, которое одновременно верит в то, что «ни один программный проект не завершается вовремя», и в существование НЛО.

Так почему же никто не составляет графиков? Есть две главные причины. Во-первых, это действительно большая морока. Во-вторых, никто не верит в их пользу. Зачем мучиться, составляя график, если все равно он окажется неверным? Сложилось общее мнение, что графики постоянно расходятся с действительностью, и чем дальше, тем значительнее, так зачем без толку мучиться?

Вот несложный способ составлять графики, которые на практике оказываются правильными.

## *1. Работайте с Microsoft Excel*

Избегайте навороченных продуктов вроде Microsoft Project. Беда в том, что его разработчики, похоже, исходили из предположения, будто вы собираетесь тратить массу времени, прослеживая зависимости. Зависимость – это такая пара задач, из которых надо завершить одну, чтобы можно было приняться за другую. По моим наблюдениям, в разработке программного обеспечения эти зависимости настолько очевидны, что не стоит тратить время на формальное слежение за ними.

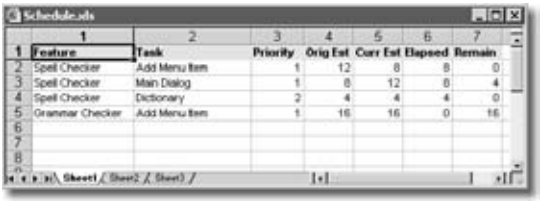
Другая неприятность состоит в том, что Project полагает, будто у вас в какой-то момент возникнет желание нажать кнопочку и «перестроить» график. В результате чего неизбежно придется перераспределять задания

[1] См. *www.jwz.org/gruntle/nomo.html*.

между исполнителями. При разработке программ это просто неразумно. Программисты не взаимозаменяемы. Джон будет исправлять ошибку Риты в семь раз дольше, чем Рита будет исправлять ошибку Риты. А если вы попробуете переключить программистку, работавшую над интерфейсом, на задачу WinSock, то она притормозит на неделю, пока не набьет руку в программировании WinSock. Вывод: Project хорош при возведении офисных зданий, а не в разработке программных проектов.

## *2. График должен быть простым*

Стандартный формат моих графиков настолько прост, что его нетрудно запомнить. Начните с семи колонок:


Schedule.xls

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | Feature | Task | Priority | Orig Est | Curr Est | Elapsed | Remain |
| 2 | Spell Checker | Add Menu Item | 1 | 12 | 8 | 8 | 0 |
| 3 | Spell Checker | Main Dialog | 1 | 8 | 12 | 8 | 4 |
| 4 | Spell Checker | Dictionary | 2 | 4 | 4 | 4 | 0 |
| 5 | Grammar Checker | Add Menu Item | 1 | 16 | 16 | 0 | 16 |
| 6 | | | | | | | |
| 7 | | | | | | | |
| 8 | | | | | | | |

Sheet1 / Sheet2 / Sheet3 /

*(1 – функция, 2 – задача, 3 – приоритет, 4 – исходная оценка, 5 – текущая оценка, 6 – потрачено времени, 7 – осталось)*

Если разработчиков несколько, заведите для каждого свой лист или добавьте колонку для имени разработчика, трудящегося над каждой задачей.

## *3. Каждая функция должна состоять из нескольких задач*

Функцией может быть, например, встроенное в программу средство проверки орфографии. Добавление этой функции состоит из нескольких отдельных задач, которые должен выполнить программист. Самое главное в составлении графика – это составить список таких задач. Отсюда важное правило:

## *4. График может составить только программист, непосредственно пишущий код*

Любая система, при которой администратор составляет график и спускает его программистам, обречена на провал. Только программист, ко-

торый будет выполнять работу, может определить, из каких этапов будет состоять реализация соответствующей функции. И только программист может оценить, сколько времени займет каждый этап.

## *5. Разбивайте задачи на возможно более мелкие части*

Это очень важно для того, чтобы график оказался реальным. Задачи должны измеряться *часами*, а не днями. (Когда я вижу график, измеряемый днями, а то и неделями, я знаю, что он нереален.) Думаете, что график с мелкими задачами просто более *детализирован*? Это не так! Если составить график, сначала грубо определив задачи, а потом разбив их на более мелкие, то получится не более точный, а *другой результат*. Итоговая цифра получается *совершенно иной*. Почему?

Когда выделяешь мелкие задачи, приходится реально определять, какие шаги надо выполнить. Написать функцию *foo*. Создать такое-то диалоговое окно. Прочесть некий файл. Длительность этих шагов правильнее оценит тот, кому уже приходилось писать функции, создавать диалоговые окна и читать файлы того или иного типа.

Если же вы поленились и выделили крупные этапы («реализовать грамматическую проверку»), значит, вы *не подумали над тем, что в действительности придется сделать*. А если вы не подумали над тем, что надо будет сделать, то вы не сможете определить, сколько это займет времени.

Из опыта известно, что задача должна занимать от 2 до 16 часов. Если в графике на задачу отводится 40 часов (неделя), значит, вы недостаточно ее раздробили.

Вот еще одно обоснование того, чтобы выбирать мелкие задачи: это заставляет вас *проектировать* функции. Если вы одним махом записали функцию под названием «интеграция с Интернетом», отведя на нее три недели, то большой вам привет, дружище. Если же вы станете определять, какие подпрограммы надо для этого написать, то функция становится вам *понятной*. Заранее осуществляя планирование на таком уровне, вы в значительной мере устраняете нестабильность в осуществлении программного проекта.

## *6. Ведите учет исходной и текущей оценок времени*

Помещая в график новую задачу, оцените, сколько часов потребуется для ее выполнения, и занесите это число в колонки исходной и теку-

щей оценок. По ходу работы, когда окажется, что задача требует больше (или меньше) времени, чем вы ожидали, можно сделать соответствующую поправку в колонке текущей оценки. Это лучший способ научиться точнее оценивать сложность задач, исходя из своих ошибок. Большинство программистов не имеют ни малейшего понятия о том, как оценивать трудоемкость задачи. Ничего страшного. Если вы будете постоянно учиться и постоянно корректировать график по мере обретения опыта, он будет действовать. (Возможно, вам придется сокращать функции или задерживать их реализацию, но график все равно будет действовать верно – в том смысле, что будет постоянно сообщать вам о необходимости либо сократить функцию, либо задержать ее реализацию.) Моя практика показывает, что большинство программистов начинает очень хорошо планировать время, потренировавшись в течение примерно года.

Когда задача завершена, в полях текущей оценки и истекшего времени будет одинаковая величина, а в поле оставшегося времени получится 0.

## *7. Ежедневно обновляйте колонку истекшего времени*

Не надо писать код с секундомером. Перед тем как идти домой или завалиться под стол спать (если вы один из *этих* гиков), предположите, что вы работали восемь часов (ха-ха!), вспомните, над какими задачами трудились, и распределите эти восемь часов по задачам в колонке истекшего времени. Значение в поле оставшегося времени Excel вычислит автоматически.

Одновременно измените значения в колонке текущей оценки для тех задач, по которым действительность внесла коррективы. *Ежедневная корректировка графика займет у вас две минуты.* Поэтому я и говорю о методе графика без хлопот – быстро и легко.

## *8. Оставьте место для отпусков, праздников и пр.*

Если ваш график охватывает примерно год, то каждый программист, наверное, возьмет 10 или 15 дней отпуска. В вашем графике должна присутствовать такая необходимая для учета функция, как отпуск, праздники и прочее, что отвлекает людей. Смысл в том, чтобы определить дату выпуска продукта, просуммировав колонку оставшегося времени и разделив на 40, – вот столько недель и остается до конца с учетом всего.

## *9. Учитывайте в графике время отладки!*

Труднее всего учитывать время отладки. Вспомните последний проект, над которым вы работали. Вполне вероятно, что отладка заняла от 100 до 200 процентов времени, потраченного на первоначальную разработку кода. Такая строка в графике должна быть, и не исключено, что это будет самая значительная из всех строк.

Вот что происходит. Допустим, разработчик трудится над функцией wawa. Исходная оценка составляла 16 часов, но потрачено уже 20 и, похоже, потребуется еще 10 часов. Поэтому разработчик вводит 30 в графу текущей оценки и 20 в графу истекшего времени.

В конце периода все «задержки» могут составить существенную величину. Теоретически, чтобы компенсировать все эти задержки, надо сократить объем реализуемых функций и за счет этого соблюсти график поставки продукта. К счастью, у нас есть эта большая и трудоемкая функция, названная Резервом, на которую уже выделено очень много часов.

В принципе разработчики отлаживают код по ходу его написания. Программист ни в коем случае не должен приниматься за новый код, пока у него есть неисправленные ошибки. Счетчик обнаруженных ошибок всегда должен быть как можно меньше по двум причинам:

1. Проще исправлять ошибки в тот же день, когда написан код. Через месяц, когда вы совершенно забудете, как работает этот код, исправлять ошибки будет очень трудно и долго.
2. Исправление ошибок напоминает научную деятельность. Невозможно предсказать, когда вы сделаете открытие и поймете, в чем ошибка. Если в каждый данный момент имеются лишь одна-две неисправленные ошибки, то нетрудно предвидеть дату окончательного выхода продукта, поскольку непредсказуемых научных изысканий остается не так много. Если же неустраненных ошибок сотни или тысячи, то нельзя предсказать, когда они будут исправлены.

Если разработчики всегда исправляют ошибки по ходу работы, то почему отладку выделяют как самостоятельный пункт? Дело в том, что даже если исправлять все ошибки сразу или в конце каждого этапа, то все равно тестеры (собственные или участники бета-тестирования) неизбежно найдут действительно *трудные* ошибки.

## *10. Учтите в графике время интеграции*

Если в вашем проекте участвует несколько программистов, то какие-то результаты работы одного из них обязательно окажутся несовместимыми с результатами работы другого и потребуется некоторая процедура согласования. Например, оба они могут без достаточных оснований поразному реализовать диалоговые окна для аналогичных задач. Кому-то придется просмотреть все меню, комбинации клавиш сокращенного вызова функций, панели инструментов и т. д., упорядочив и причесав все пункты меню, добавленные произвольным образом. Когда два человека вводят в систему свой код, появляются ошибки компиляции, которые надо исправлять. Для этого надо предусмотреть отдельную строку в графике.

## *11. Оставляйте в графике резерв*

Работа имеет тенденцию затягиваться сверх сроков. Есть два вида резервов, которые имеет смысл принимать во внимание. Во-первых, это резерв для задач, потребовавших больше времени, чем первоначально предполагалось. Во-вторых, это резерв для задач, выполнение которых изначально не предполагалось, поскольку руководство решило, что реализация чего-то этакого КРАЙНЕ ВАЖНА и не может быть отложена до следующей версии.

Возможно, вы удивитесь, что отпуска, праздники, интеграция и резерв в сумме составят больше, чем действительные задачи. Если вас это удивляет, то вы, вероятно, не очень давно программируете. Можете и не учитывать их – на свой страх и риск.

## *12. Ни в коем случае не позволяйте программистам занижать оценку времени*

Многие неопытные менеджеры полагают, что могут «стимулировать» более быструю работу своих программистов, если будут задавать для них трудные, «напряженные» (нереалистично быстрые) графики. Я думаю, что такого рода мотивация – глупость. Если я выбиваюсь из графика, я чувствую себя обреченным, подавленным и немотивированным. Если же я работаю с *опережением* графика, я чувствую себя бодрым и работоспособным. Психологические игры с графиком очень опасны.

Если ваш руководитель требует, чтобы вы занизили свою оценку, то заведите в графике еще одну колонку и назовите ее «Оценка Рика» (если вас зовут Рик). Запишите в нее свою оценку. Пусть ваш руководитель делает с колонкой «текущая оценка» все, что ему заблагорассудится. Не обращайте внимания на его оценки. Когда проект завершится, посмотрите, кто из вас оказался ближе к действительности. Я обнаружил, что одна лишь *угроза* сделать это творит чудеса, особенно если ваш руководитель поймет, что теперь он вынужден смотреть, как вы будете доказывать, насколько *медленно* вы можете работать!

Почему неумелые менеджеры стараются заставить программистов сократить свои оценки?

В начале проекта действуют технические менеджеры, которые встречаются с людьми бизнеса и представляют список функций, которые, как им *кажется*, можно сделать за три месяца и которые на самом деле обернутся девятью. Думая о написании кода и не вникая при этом во все шаги, которые придется выполнить, вы всегда будете приходить к выводу, что потребуется время $n$, тогда как в реальности оно, возможно, окажется ближе к $3n$. Составляя реальный график, вы суммируете время всех задач и обнаруживаете, что проект займет гораздо больше времени, чем первоначально предполагалось. Жизнь диктует свои законы.

Неопытные менеджеры пытаются бороться с этим, заставляя людей работать быстрее. Это нереалистичный подход. Можно нанять дополнительных сотрудников, но им потребуется время, чтобы начать работать с полной эффективностью, и не исключено, что несколько месяцев они будут работать вполсилы (снижая при этом эффективность тех, кто станет их обучать). В любом случае полгода уйдет на то, чтобы заполучить хороших программистов, – таков уж этот рынок.

Вы можете *временно* получить от своих сотрудников на 10 процентов больше сырого кода за счет того, что на 100-процентную производительность труда они выйдут лишь через год. Не слишком большая прибавка, к тому же это напоминает проедание семенного фонда.

Можно попытаться получить на 20 процентов больше сырого кода, упросив всех трудиться изо всех сил, не взирая на усталость. Бац – и время отладки *удвоилось*. Идиотский шаг, который приведет к весьма показательным результатам.

Но нельзя получить $3n$ из $n$, ни за что. А если вам кажется, что вы сможете это сделать, то сообщите мне, пожалуйста, код акций вашей компании на бирже, и я с успехом сыграю на понижение.

## *13. График похож на набор кубиков*

Если у вас есть кучка деревянных кубиков, которые не помещаются в коробку, есть два выхода: взять коробку побольше или убрать некоторые кубики. Если вы думали, что сможете сделать продукт через 6 месяцев, а по графику получается 12, вам придется либо отложить выпуск продукта, либо отказаться от каких-то функций. Сделать кубики меньше вы не можете, а если вы считаете, что можете, значит, вы лишаете себя полезной возможности реально *смотреть в будущее* и лжете себе о том, что вы там видите.

Между прочим, еще одно замечательное свойство такого ведения графика состоит в том, что *приходится* отказываться от функций. Что в этом хорошего? Представьте себе, что есть две функции, одна из которых действительно полезна и значительно повышает ценность вашего продукта (пример: таблицы в Netscape 2.0), а другая действительно проста и программисты охотно ее напишут (пример: тег `BLINK`), но не имеет какой-либо практической или маркетинговой ценности.

Если вы не составите график, то программисты сначала сделают ту функцию, которую написать легко или интересно. Потом у них не будет хватать времени, и вам придется продлевать график, чтобы все-таки сделать полезную/важную функцию.

Если же у вас будет график, то еще до начала работы вы поймете, что надо что-нибудь сократить, поэтому уберете легкую/интересную функцию и оставите полезную/важную. Заставив себя отказаться от некоторых функций, вы получите в итоге более мощный и удачный продукт с лучшим набором функций, который и готов будет раньше.

Вспоминаю свою работу над Excel 5. Изначальный перечень функций был огромен и вышел бы *далеко* за рамки графика. Мы очень переживали. *Все* эти функции были такими важными! Разве можно было обойтись без помощника редактирования макросов?

Но выбора у нас не было, и мы срезали все лишнее «до костей», как нам казалось, чтобы уложиться в график. Все переживали из-за сокращений. Чтобы успокоить себя, мы решили, что *не отказываемся* от этих функций, а *откладываем их* до Excel 6, поскольку они менее важны.

Когда работа над Excel 5 близилась к завершению, я вместе со своим коллегой Эриком Мишельманом приступил к спецификации Excel 6. Мы решили просмотреть список функций «Excel 6», которые были вырезаны из графика Excel 5. К нашему полному *изумлению,* он оказался самым дрянным списком функций, который можно себе представить. *Ни одна* из

этих функций не заслуживала того, чтобы ей занимались. По-моему, ни одну из них не реализовали даже в трех последовавших затем релизах. Процедура отбора функций с целью уложиться в график оказалась лучшим, что мы могли сделать. В ином случае Excel 5 делалась бы вдвое дольше и наполовину состояла бы из ненужного барахла. (Я абсолютно уверен, что именно это происходит с Netscape 5/Mozilla: у них нет графика, у них нет полного списка функций, никто не хочет поступиться никакими функциями, и они не выдали готовой версии. Когда они выпустят готовую версию, в ней будет масса ненужных функций типа IRC-клиентов, на которые не стоило тратить время.)

### *Что вам следует знать об Excel*

Одна из причин, по которым Excel так замечательно помогает в составлении графиков написания программ, заключается в том, что большинство разработчиков Excel используют ее с единственной целью, а именно для составления своих графиков написания программ! (Очень немногие из них пользуются бизнес-сценариями «what-if» – в конце концов, здесь же программисты собрались!)

**Листы с совместным доступом.** Команда File→Shared Lists (или Tools→Share Workbook в новых версиях) позволяет всем одновременно открыть один и тот же файл и редактировать его. Поскольку вся команда должна постоянно корректировать график, это действительно полезная возможность.

**Автофильтрация.** Это отличный способ фильтрации графика, позволяющий, например, посмотреть только те функции, которые закреплены за вами лично. Сочетая его с автоматической сортировкой, можно получить список всех назначенных вам заданий в порядке их приоритетов. Круто!

**Сводные таблицы.** Прекрасное средство посмотреть итоги и кросс-таблицы. Можно, например, построить диаграмму оставшегося времени работы для каждого разработчика и каждого приоритета. Вам необходимо научиться пользоваться сводными таблицами, потому что они увеличивают мощь Excel в тысячу раз. Сводные таблицы – это как ломтики хлеба, намазанные шоколадным маслом.

**Функция WORKDAY.** Входящая в состав аналитического пакета Excel, функция WORKDAY позволяет получить из графика календарные даты.

ГЛАВА ДЕСЯТАЯ

# Ежедневная сборка – лучший друг программистов

27 января 2001 года, суббота

В 1982 году в моей семье был получен самый первый в Израиле IBM-PC. Мы поехали на склад и ждали там, пока наш PC привезут из порта. Каким-то образом мне удалось подбить отца на полностью укомплектованный вариант с *двумя* флоппи-дисками, 128 килобайтами памяти и двумя принтерами – матричным (для быстрой черновой печати) и лепестковым принтером Brother для качественной печати, который во время работы издавал такой же звук, как автоматическая винтовка, только громче. По-моему, мы получили все мыслимые аксессуары: PC-DOS 1.0, техническое руководство за 75 долларов с полной распечаткой исходного кода BIOS, макроассемблером и потрясающим монохромным дисплеем IBM с 80 колонками и даже буквами нижнего регистра! Все это стоило около $10 000, включая смехотворные по нынешним временам израильские налоги на импорт. Полный восторг!

Кроме того, «всем» было известно, что BASIC – детский язык, вынуждающий писать «макаронный» код и превращающий мозги в сыр камамбер. Поэтому мы выложили 600 долларов за IBM Pascal, который поставлялся на трех дискетах. Первый проход компилятора был на первой дискете, второй проход – на второй, а компоновщик располагался на третьей. Я написал простую программу «hello, world» и скомпилировал ее. Общее время работы: 8 минут.

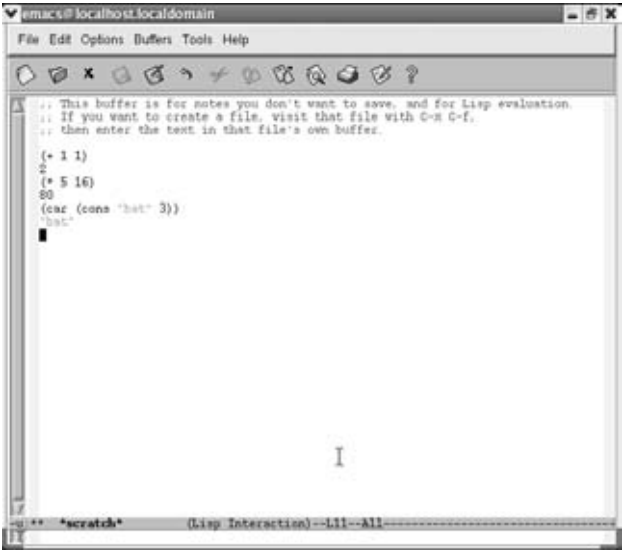
Н-да. Многовато. Я написал пакетный файл для автоматизации работы и сократил время до 7,5 минут. Уже лучше. Но когда я попытался писать длинные программы, типа моей потрясающей версии «Отелло», *всегда* меня побивавшей, основное время стало уходить на ожидание результатов компиляции. «Ну! – сказал мне знакомый профессиональный програм-

мист. – Мы принесли на работу гимнастическую скамью для пресса и занимались во время компиляции сед-гимнастикой. Пара месяцев программирования, и у меня был железный брюшной пресс.»

В один прекрасный день я познакомился с классной программой под названием Compas Pascal, которую Филип Канн (т. е. Borland) купил у датской фирмы POLY DATA и переименовал в Borland Turbo Pascal.[1] Turbo Pascal просто потрясал, потому что делал практически то же, что IBM Pascal, но занимал примерно 33 килобайта *вместе с текстовым редактором*. Это было просто изумительно. Еще больше поражало, что небольшую программу можно было скомпилировать быстрее чем за 1 секунду. Это как если бы никому не известная компания стала выпускать клон Buick LeSabre, двигающийся со скоростью миллион километров в час и объезжающий земной шар на таком количестве бензина, которого муравью не хватит, чтобы нанюхаться.

Внезапно моя производительность *резко* возросла.

Вот тогда я узнал про понятие *REP-цикла*. REP означает «Read, Eval, Print» и описывает то, чем зарабатывает себе на жизнь интерпретатор Lisp: он читает входные данные, производит на их основе вычисления и выводит результат. На этом скриншоте приведен пример REP-цикла: я что-то печатаю, интерпретатор Lisp это читает, вычисляет и печатает результат.




[1] См. *community.borland.com/article/0,1410,20161,00.html*.

Если рассмотреть процесс в более крупном масштабе, то можно сказать, что когда вы пишете код, то находитесь в макро-версии REP-цикла, называемой «циклом редактирования–компиляции–тестирования». Вы редактируете код, компилируете его и тестируете, проверяя, хорошо ли он работает.

Важно отметить, что при написании программы этот цикл надо пройти многократно, поэтому чем быстрее выполняется цикл редактирования–компиляции–тестирования, тем выше ваша производительность, и ограничивает ее природная невозможность мгновенной компиляции. Это формальная теоретическая причина, по которой программистам нужны *как можно более быстрые компьютеры*, а разработчики компиляторов прилагают всяческие усилия для достижения сверхкороткого цикла редактирования–компиляции–тестирования. В Visual Basic этому способствует выполнение грамматического и синтаксического анализа каждой строки по мере ее ввода, благодаря чему окончательная компиляция оказывается сверхбыстрой. В Visual C++ для этого есть инкрементная компиляция, прекомпилированные заголовки и инкрементная сборка.

Но как только вы начинаете работать в большой команде, где много разработчиков и тестеров, вы снова сталкиваетесь с тем же самым циклом, только гораздо более крупным (вот они где – фракталы!). Тестер находит в коде ошибку и регистрирует ее. Программист исправляет ошибку. Через какое время тестер получит исправленный вариант кода? В некоторых организациях этот цикл «тест–регистрация ошибки–исправление–повторное тестирование» может занять пару недель, что свидетельствует о непродуктивности работы организации в целом. Для того чтобы весь процесс разработки протекал без сбоев, необходимо сократить цикл между регистрацией ошибки и повторным тестированием.

Для этого целесообразно применять *ежедневные сборки*. Ежедневная сборка – это *автоматическая ежедневная полная сборка* всего дерева исходного кода.

**Автоматическая**. Компиляция кода запускается в фиксированное время суток с помощью задания cron (в UNIX) или службы планировщика (под Windows).

**Ежедневная** (или более частая). Желательно было бы проводить сборку непрерывно, но это может оказаться невозможным из-за проблем контроля версий исходного кода, о которых я скажу чуть позже.

**Полная**. Не исключено, что ваш код существует в нескольких версиях: для разных языков, разных операционных систем или разной комплек-

тации. При ежедневной сборке надо компилировать их *все*. И необходимо выполнять компиляцию всех файлов с самого начала, не полагаясь на средства инкрементной сборки, которые могут быть несовершенны.

Вот некоторые преимущества, обеспечиваемые ежедневной сборкой:

1. После исправления ошибки тестеры быстро получают новую версию и могут снова проверить, действительно ли исправлена ошибка.
2. Разработчики получают уверенность в том, что их изменения в коде не нарушат работу ни одной из 1024 версий выпускаемой системы, и при этом им *не надо ставить* на свой стол машину с OS/2, чтобы на ней тестировать.
3. Разработчики, вносящие свои модификации непосредственно перед плановой ежедневной сборкой, знают, что не нарушат работу всех остальных, если сделают что-то, из-за чего сборка «рухнет» и *никто* не сможет выполнить компиляцию. Это эквивалент «голубого экрана смерти» для целого программистского коллектива, который часто возникает, если программист забывает добавить в хранилище созданный им новый файл. В этом случае на *его собственной* машине сборка проходит прекрасно, но у всех остальных возникают ошибки компоновщика, и всякая работа прекращается.
4. Внешние группы, такие как маркетинг, клиенты с бета-версиями и пр., работающие с незаконченным продуктом, могут взять ту сборку, которая достаточно устойчива, и некоторое время пользоваться ею.
5. Обнаружив новую неожиданную ошибку, причина которой непонятна, можно путем бинарного поиска в этом архиве найти ту сборку, в которой эта ошибка появилась впервые (конечно, если вы ведете архив ежедневных сборок). При хорошем контроле версий исходного кода удается даже определить автора кода, содержащего ошибку.
6. Когда тестер сообщает об ошибке, которую программист считал исправленной, тестер может уточнить, в какой сборке ее обнаружил. Тогда программист проверяет дату исправления в дереве кода и была ли ошибка исправлена *в действительности*.

Вот как надо делать ежедневную сборку. В первую очередь для ежедневной сборки нужен сервер, и лучше, если удастся выделить для него самую быструю из имеющихся машин. Напишите сценарий, который получит полный экземпляр текущего кода из хранилища (или вы *не пользуетесь* системой контроля версий?), а затем скомпилирует, с самого начала, все версии кода, которые вы поставляете. Если есть программа инсталляции,

соберите ее тоже. В ежедневной сборке должно участвовать все, что вы поставляете клиентам. Поместите каждую сборку в отдельный каталог с указанием даты. Запускайте этот сценарий в один и тот же час ежедневно.

Вот некоторые советы по ежедневной сборке.

- Важно, чтобы сценарий ежедневной сборки делал *все*, что требуется для окончательной сборки: от загрузки кода из хранилища до помещения результатов на веб-сервер для открытого доступа (хотя в период разработки это, конечно, должен быть тестовый сервер). Только так можно гарантировать, что *какой-то* процесс сборки не останется «документированным» только в голове одного человека. И вы никогда не окажетесь в ситуации, когда не можете выпустить продукт только потому, что один Вася знает, как создавать инсталлятор, а Вася вчера попал под автобус. В команде Juno для того чтобы выполнить полную сборку с самого начала, достаточно было знать, где находится сервер сборки и как щелкнуть по значку «ежедневная сборка».
- Проще всего лишиться рассудка в ситуации, когда надо выпускать продукт, а *одна маленькая ошибочка* так и осталась неисправленной, поэтому ошибочку вы исправите на сервере, предназначенном для ежедневной сборки, и продукт будет готов к поставке. Железное правило: поставлять можно только код, полученный в результате безошибочной полной сборки, произведенной из полного исходного кода.
- Установите максимальный уровень вывода предупреждений компиляторов (-W4 для продуктов Microsoft; -Wall для gcc) и необходимость остановки при любом предупреждении.
- Если дневная сборка не прошла, то может остановиться вся работа. Бросьте все и повторяйте сборку, пока ошибка не будет исправлена. В некоторые дни приходится выполнять по несколько сборок.
- Сценарий ежедневной сборки должен с помощью электронной почты оповещать всех разработчиков об ошибках. Не так сложно grep-нуть в логах «error» или «warning» и включить результат в почтовое сообщение. Сценарий может также помещать отчет о состоянии сборки на общедоступную страницу HTML, чтобы программисты и тестеры могли быстро узнать, какие сборки завершились успехом.
- В команде разработчиков Microsoft Excel мы с большим успехом следовали правилу, согласно которому тот, по чьей вине не прошла сборка, контролировал прохождение всех последующих сборок, пока кто-

нибудь следующий не заваливал сборку. Это служило хорошим стимулом не срывать сборку, а также позволяло почти каждому пройти через работу ответственного за сборку, и в итоге все умели это делать.

- Если вся группа работает в одном часовом поясе, то удобно выполнять сборку в обеденный перерыв. В этом случае все помещают свой самый свежий код в хранилище перед самым обедом, сборка проходит, пока все едят, а когда все возвращаются и оказывается, что сборка не прошла, то все, кто может исправить положение, оказываются на месте. Как только сборка пройдет, все могут загрузить себе последнюю версию, не опасаясь, что работа остановится из-за неуспешной сборки.
- Если группа работает в двух часовых поясах, спланируйте ежедневную сборку так, чтобы люди в одном часовом поясе не останавливали работу людей в другом часовом поясе. В команде Juno те, кто работал в Нью-Йорке, загружали код в хранилище в 7 вечера местного времени и отправлялись домой. Если они ломали при этом сборку, то хайдарабадская команда в Индии приходила на работу (примерно в 8 вечера по нью-йоркскому времени) и была парализована на весь день. Мы стали проводить две сборки в день, примерно за час до ухода домой каждой бригады, и полностью решили эту проблему.

Вот некоторые другие ресурсы по данной теме:

- Небольшое обсуждение инструментов для ежедневной сборки есть в сети по адресу: *discussfogcreek.com/joelonsoftware/default.asp?cmd=show&ixPost=862*.
- Ежедневное проведение сборки настолько важно, что это первый из 12 приемов получения лучшего кода, перечисляемых в главе 3.
- Много интересной информации о проведении сборок (еженедельных) командой Windows NT в книге Дж. Паскаль Захари «Showstopper».[1]
- Стив Макконнелл (Steve McConnell) пишет о ежедневных сборках на своем веб-сайте *www.construx.com/stevemcc/bp04.htm*.

[1] Zachary, Pascal G. (Дж. Паскаль Захари) «Showstopper! The Breakneck Race to Create Windows NT and the Next Generation at Microsoft» (Microsoft лихорадочно спешит создать Windows NT и Next Generation), The Free Press, 1994.

ГЛАВА ОДИННАДЦАТАЯ

# Тотальное уничтожение ошибок

31 ИЮЛЯ 2001 ГОДА, ВТОРНИК

Качество программного обеспечения, а точнее отсутствие оного вызывает жалобы у всех. Теперь у меня своя собственная компания, и я вынужден как-то решать эту проблему. В последние две недели мы в Fog Creek занимались только подготовкой к выпуску новой инкрементной версии FogBUGZ с намерением устранить все известные ошибки (числом около 30).

Для разработчика программного обеспечения исправление ошибок – это дело правое. Так? Ведь это всегда правое дело?

Нет!

Исправлять ошибки необходимо только тогда, когда сохранение ошибки обходится дороже, чем ее исправление.

Измерить стоимость того и другого непросто, но можно. Приведу пример. Допустим, что вы управляете фабрикой по производству сэндвичей с арахисовым маслом и желе. Ваша фабрика выпускает 100 000 сэндвичей в сутки. Недавно благодаря появлению некоторых новых сортов (чесночно-арахисовое масло с острым соусом хабанеро) спрос на вашу продукцию поднялся выше крыши. Выпуская 100 000 сэндвичей, фабрика работает на полную мощность, но спрос, вероятно, ближе к 200 000. Вы просто не в состоянии произвести больше. Каждый сэндвич дает вам прибыль в 15 центов. Стало быть, вы ежедневно теряете 15 000 долларов потенциальной прибыли из-за отсутствия достаточных мощностей.

Строить еще одну фабрику – слишком дорого. У вас нет такого капитала, и вы опасаетесь, что мода на пряные/чесночные сэндвичи может закончиться. Но вы продолжаете терять по 15 000 в день.

К счастью, вы взяли на работу Джейсона. Джейсон – программист 14-ти лет от роду, который поковырялся в компьютерах, управляющих производством, и теперь ему кажется, что он знает, как увеличить скорость конвейера вдвое. Начитался на Slashdot про разгон чипов. Проведенное испытание показало, что вроде бы это возможно.

Есть только одно обстоятельство, удерживающее вас от того, чтобы дать делу полный ход. Из-за какой-то *крохотной,* ничтожной, жалкой ошибочки примерно раз в час производственная линия заминает один сэндвич. Джейсон хочет исправить эту мелкую ошибку. Он считает, что ему будет достаточно трех дней. Что вы сделаете – дадите ему возможность исправить ошибку или запустите программу в теперешнем состоянии, с известной ошибкой?

Откладывание запуска программы на три дня обойдется в 45 000 долларов упущенной прибыли. А ваша экономия составит стоимость сырья для 72 сэндвичей. (Джейсон в любом случае исправит ошибку через 3 дня.) Ну, не знаю, сколько стоят сэндвичи *на вашей* планете, но здесь, на Земле, они намного дешевле 625 долларов.

Так о чем это я говорил? Ах, да. Иногда *исправлять ошибку невыгодно.* Вот другой пример: в вашей программе есть ошибка, приводящая к ее аварийному завершению при открытии очень больших файлов, но она может возникнуть лишь у единственного пользователя, который работает в OS/2 и у которого, как вам известно, больших файлов не бывает. Ну и нечего ее исправлять. Бывают вещи и похуже. Еще я перестал думать о тех, у кого мониторы с 16 цветами или кто когда-то поставил Windows 95 и не обновлял ее 7 лет. Можете мне поверить, эти люди не станут тратить деньги на коробочные программные продукты.

Но обычно ошибки исправлять надо. Даже «безвредные» ошибки портят репутацию компании и продукта, которая в конечном итоге сильно повлияет на ваши доходы. Избавиться от репутации компании, выпускающей продукт с ошибками, трудно. Если вам все же надо выпустить продукт версии .01, то вот некоторые соображения о том, как искать и исправлять «*правильные*» ошибки, т.е. такие, исправление которых экономически оправданно.

## *1. Обеспечьте получение информации об ошибках*

В случае FogBUGZ мы прибегли для этого к двум способам. Во-первых, мы перехватываем все ошибки на нашем бесплатном демосервере, со-

бираем как можно больше информации о них и отправляем все это электронной почтой бригаде разработчиков. В результате мы обнаружили массу ошибок, что было очень хорошо. Например, мы выяснили, что ряд людей не вводит дату в нужном месте экрана «Fix For». Тут даже не было сообщения об ошибке, просто программа аварийно завершалась (что для веб-приложения означает получение гадкого сообщения IIS вместо ожидаемой страницы).

Когда я работал с Juno, у нас была еще более замечательная система автоматического получения данных ошибок «прямо с поля боя». С помощью TOOLHELP.DLL мы установили обработчик, который при каждом аварийном завершении Juno не умирал, пока стек сбрасывался в log-файл, и только потом сходил в могилу. Когда программа в очередной раз соединялась с Интернетом, чтобы отправить почту, она загружала log-файл. На стадии бета-тестирования мы собирали эти файлы, сравнивали все аварийные завершения и вводили их в базу данных ошибок. В результате обнаружились буквально сотни ошибок, вызывающих аварийное завершение. Когда у вас миллион пользователей, можно придти в *полное изумление* от того, какой именно код может приводить к аварии, часто из-за очень малого объема доступной памяти или очень плохих компьютеров (вы напишете Packard Bell без ошибок?). Например, в коде

```
int foo( object& r )
{
   r.Blah();
   return 1;
}
```

возникал фатальный сбой из-за того, что ссылка r оказывалась равной NULL, хотя это совершенно невозможно – *не бывает* нулевых ссылок, C++ это гарантирует. Можете мне не верить, но если, имея миллионы пользователей, подождать некоторое время и скрупулезно собирать дампы их стеков, то обнаруживаются фатальные сбои в таком вот коде, вы их видите, но не верите своим глазам. (Их нельзя исправить, потому что в них виноваты космические лучи. Купите новый компьютер и больше *не* устанавливайте в панель задач никаких красивых шароварных штуковин, которые вам попадутся.)

Кроме того, абсолютно все обращения в службу технической поддержки проверяются на предмет, не связаны ли они с наличием ошибки. Приняв вызов, мы пытаемся определить, что можно сделать для его аннулирования. Например, в прежней Setup для FogBUGZ предполагалось, что Fog-

BUGZ будет выполняться под учетной записью анонимного пользователя Интернета. В 95 процентах случаев это было оправданно, а в 5 процентах предположение было неверным, но каждый из этих 5 процентов случаев заканчивался звонком в нашу службу поддержки. Поэтому мы модифицировали Setup, чтобы он запрашивал имя учетной записи.

## *2. Обеспечьте себе экономичную обратную связь*

Не всегда можно *точно* определить, во что обойдется исправление каждой ошибки, но кое-что сделать *можно*: предъявить счет на «стоимость» технической поддержки производственному подразделению. В первой половине 1990-х в Microsoft началась финансовая реорганизация, в результате которой каждое производственное подразделение должно было возмещать полную стоимость всех обращений за технической поддержкой. В результате производственные подразделения стали настаивать, чтобы PSS (служба технической поддержки Microsoft) регулярно предоставляла им «горячую десятку» ошибок. Когда разработчики сосредоточились именно на этих ошибках, стоимость поддержки продуктов резко упала.

Это несколько противоречит современной тенденции, когда отдел технической поддержки оплачивает собственное функционирование, как это происходит в большинстве крупных компаний. В Juno предполагалось, что техническая поддержка будет содержать себя, взимая плату с клиентов. Перемещая экономическую ответственность за ошибки на самих пользователей, вы лишаетесь даже ограниченных возможностей определить наносимый им ущерб. (Взамен вы получаете разгневанных пользователей, возмущенных тем, что они должны платить за *ваши* ошибки, и рассказывающих об этом своим друзьям, так что трудно сказать, во сколько это обойдется вам в итоге. Справедливости ради надо сказать, что собственно Juno был бесплатным продуктом, поэтому перестаньте ругаться.)

Способ примирить одно с другим состоит в том, чтобы *не* взимать с пользователя денег, если обращение в службу поддержки было вызвано ошибкой в вашем продукте. Microsoft так и делает, и это правильно, и я ни разу не платил за звонок в Microsoft! Зато счет на 245 долларов, или сколько там стоит в наше время один случай, предъявляется производственному отделу. В результате их прибыль от проданного вам продукта улетучивается, что создает совершенно правильные экономические стимулы. Это напоминает мне, почему игры для DOS были *жутким* бизнесом... Чтобы они прилично выглядели и быстро работали, обычно требовались необычные

видеодрайверы, и одно-единственное обращение в службу поддержки по поводу видеодрайверов могло съесть прибыль от *20* экземпляров вашего продукта, если Egghead, Ingram и то объявление на MTV еще не поглотили *всех* ваших накоплений.

## *3. Посчитайте, во что вам обойдется исправить их все*

Наша Fog Creek Software – крохотная компания (конечно, мысленно мы представляем себе ее иначе), и команда разработчиков сама же принимает все обращения за технической поддержкой. Это обходилось нам примерно в час времени ежедневно и в соответствии с нашими тарифами на консультирование стоило примерно 75 000 долларов в год. Нам казалось, что мы сможем уменьшить это время до 15 минут в день, если исправим все известные ошибки.

По очень грубым прикидкам это означает, что чистая приведенная экономия составит примерно 150 000 долларов. Это оправдывает 62 дня работы; если вы сможете сделать это быстрее, чем за 62 человеко-дня, стоит браться за работу.

С помощью удобной функции оценки, имеющейся в FogBUGZ, мы подсчитали, что исправить все ошибки можно за 20 человеко-дней (двое сотрудников будут работать две недели), т. е. «потратив» 48 000 и получив взамен 150 000, и это будет очень неплохая инвестиционная прибыль *благодаря экономии на одной только технической поддержке*. (Заметьте, что вместо тарифа на консультационные услуги можно было взять величину зарплат программистов и накладных расходов и получить тот же результат 3:1, поэтому они равны.)

Я даже не стал считать прибыль от получения более совершенного продукта, но можно сделать и это. Старый код в течение одного месяца июля дал нам 55 аварий на демо-сервере, у 17 разных пользователей. Можно представить себе, что хотя бы *один* из них решил не покупать FogBUGZ из-за ошибок при работе демо-версии (точной статистики у меня нет). Во всяком случае, объем потерянных продаж мог составить от 7000 до 100 000 долларов в приведенном значении. (Если заняться этим серьезно, то можно получить и реальные цифры.)

Еще один вопрос. Можно ли назначать более высокую цену за продукт на основании того, что в нем меньше ошибок? Это еще больше увеличило бы ценность устранения ошибок. Мне кажется, что в исключительных слу-

чаях количество ошибок влияет на цену, но мне затруднительно привести в подтверждение пример из области коробочных продуктов.

## *Только не бейте!*

Кто-то, прочитав эссе вроде этого, несомненно придет к нелепым выводам типа «Джоэл не советует исправлять ошибки». На самом деле я считаю, что в большинстве случаев исправление ошибок экономически оправданно. Но иногда можно получить более высокую прибыль *иными* способами, не занимаясь тотальным истреблением ошибок. Если приходится выбирать между исправлением ошибки для того малого с OS/2 и добавлением новой функции, благодаря которой можно продать 20 000 экземпляров программы в General Electric, то тогда, друг с OS/2, извини. Если же упрямство *все-таки* заставляет тебя считать, что исправить ошибку с OS/2 важнее, чем добавить функцию для GE, то не исключено, что конкуренты, которые думают иначе, вытеснят тебя с рынка.

Тем не менее в душе я остаюсь оптимистом и верю, что выпуск продукции очень высокого качества таит в себе получение высокой прибыли, которую не так просто оценить. Ваши сотрудники будут больше гордиться своим трудом. Меньше ваших клиентов вернет вам обратно по почте CD с вашим продуктом, предварительно изжарив его в микроволновке и порубив топором. Поэтому я склоняюсь к качеству (мы действительно исправили в FogBUGZ *все известные ошибки*, а не только самые заметные) и уверен, что благодаря полному устранению ошибок с демосервера мы создали чрезвычайно надежный продукт.

ГЛАВА ДВЕНАДЦАТАЯ

# Пять миров

6 мая 2002 года, понедельник

Первым местом моей постоянной работы был хлебозавод. В одном огромном помещении пекли хлеб, а в другом упаковывали его в коробки. В первом целый день все мучились с тестом. Оно застревало в механизмах, прилипало к рукам, волосам и обуви, и у каждого был маленький скребок для краски, которым это тесто соскабливали. Во втором помещении все целый день мучились с крошками: они застревали в механизмах и в волосах, и у каждого была маленькая кисточка. Мне подумалось, что в каждой работе есть свое зло – собственный источник вечных неприятностей, и я порадовался, что не работал на фабрике бритвенных лезвий.

В программировании есть свои источники зла, и в одном помещении они не такие, как в другом, но почему-то какую бы книгу о разработке программного обеспечения вы ни взяли, вряд ли найдете в ней упоминания о разных помещениях, не говоря уже о существующих разновидностях зла.

Большую часть своей послехлебозаводской профессиональной деятельности я занимался созданием программного обеспечения, предназначенного для использования миллионами людей. Когда я впервые прочел, что в «экстремальном программировании» считается обязательным иметь в составе бригады «клиента», я пришел в недоумение. При работе над почтовым клиентом для Juno, крупного общенационального интернет-провайдера, у нас были *миллионы* клиентов, включая великое множество милых старушек, о которых ничего не говорится в литературе, посвященной формированию команд программистов.

Когда мне доводилось выступать с лекциями перед корпоративными программистами и я начинал распространяться перед ними о том, что у нас в Microsoft было так-то и так-то, они смотрели на меня, как на инопланетянина: «Джоэл, здесь не Microsoft, у нас *нет* миллионов клиентов, у нас только Джим во вспомогательном офисе.»

В конце концов я разобрался. Существуют разные сферы разработки программ, и в каждой из них действуют свои законы. А этот кажущийся очевидным факт долго игнорировался. Да, все мы разработчики программного обеспечения, но когда мы не обращаем внимания на различия между нашими сферами, начинаются неуместные войны .NET против Java или методологий ускоренной разработки против программных технологий. Вы читаете статью о моделировании UML, но там нигде не сказано, что оно бессмысленно при программировании драйверов устройств. Или вы читаете в телеконференции, что 20-мегабайтный исполняемый модуль .NET не представляет никаких проблем, но там не сказано, что если надо написать код для пейджера с 32 килобайтами EPROM, то это уже проблема, да еще какая.

Я считаю, что можно выделить пять сфер:

- Коробочные продукты
- Внутрифирменное ПО
- Встроенное ПО
- Игры
- Одноразовые программы

## *1. Коробочные продукты*

Я называю так (shrinkwrap) все продукты, с которыми будут «самостоятельно» работать многие люди. Эти продукты могут быть упакованы в целлофан и продаваться в CompUSA либо загружаться через Интернет. Эти программы могут быть коммерческими, условно-бесплатными (shareware), с открытым кодом, GNU или с какими-либо другими условиями распространения; главное в том, что их установят и будут использовать тысячи и миллионы людей. Возможно даже, что это веб-сайт, посещаемый тысячами или миллионами людей.

Жизнь разработчиков коробочных продуктов определяется двумя вечными неприятными факторами:

- У их программ неисчислимое количество пользователей, часто имеющих альтернативы. Поэтому для успеха программы интерфейс пользователя должен быть проще, чем обычно. Половину времени они тратят на то, чтобы *еще больше упростить* жизнь своей деревенской тетушке.
- Их программы выполняются на многих компьютерах, поэтому код должен быть предельно устойчивым к различиям между машинами. Обеспечив это требование, они тратят *оставшееся* время на то, чтобы разглядеть метки версий DLL.

На прошлой неделе я получил письмо об ошибке в продукте CityDesk, выпускаемом моей компанией. Эта ошибка проявляется только с польскими клавиатурами, потому что на них для ввода специальных символов применяется так называемая клавиша AltGr (никогда об этом не слышал). Мы тестировали Windows 95, 95OSR2, 98, 98SE, Me, NT 4.0, Windows 2000, и Windows XP Home и Pro. Мы тестировали с установленными IE 5.01, 5.5 или 6.0. Мы тестировали версии Windows, локализованные для английского, испанского, французского, иврита и традиционного китайского языков. Но мы еще *не совсем* добрались до польского.

Есть три главные разновидности коробочного продукта: с открытым исходным кодом (open source), веб-приложения и консалтинговое ПО.

### *Приложения с открытым исходным кодом*

Эти программы часто разрабатываются бесплатно, что сильно влияет на динамику. Например, в команде, состоящей исключительно из добровольцев, часто не делают то, что считается «неинтересным», что, как красноречиво объяснил Мэттью Томас (Matthew Thomas), наносит явный ущерб юзабилити.[1] Разработчики очень часто географически удалены друг от друга, что совершенно меняет характер обмена информацией между ними. В мире open source очень редко происходит личное общение, когда рисуют прямоугольники и стрелки на доске, поэтому проектные решения, для которых полезно вычерчивать квадратики и стрелки, в таких проектах удаются обычно плохо. В итоге географически распределенные команды гораздо больше преуспевают в клонировании уже готового программного обеспечения, когда проектирование если и требуется, то в незначитель-

[1] Matthew, Thomas (Мэттью, Томас) «Why Free Software Usability Tends to Suck» (Почему тошнит от юзабилити бесплатного ПО), 2002, *http://mpt. phrasewise.com/2002/04/13.*

ном объеме. Но по большей части программное обеспечение все же рассчитано на самостоятельное выполнение всеми желающими, поэтому я причисляю его к коробочному.

### *Веб-приложения*

Такое программное обеспечение, как Hotmail, eBay или даже контентный сайт, тоже должно быть простым в использовании и работать во многих броузерах. У большинства пользователей установлен «броузер-монополист», но больше всего шума от пользователей, у которых по неизвестным причинам стоит какой-нибудь доисторический броузер, не способный правильно показывать соответствующий стандартам HTML, даже если в приложение *встроить* самого Джеффри Зельдмана. Поэтому в итоге масса времени тратится на поиск в Google таких вещей, как «ошибка шрифта CSS в Netscape 4.72». Хотя в данном случае у разработчиков есть возможность хотя бы частично контролировать среду «развертывания» – компьютеры в центре обработки данных, но им приходится иметь дело с большим разнообразием броузеров и большим количеством пользователей, поэтому я считаю, что по существу это разновидность коробочного продукта.

### *Консалтинговое ПО*

Эта разновидность коробочного продукта требует такого объема работы при настройке и установке, что его запуск должна обеспечивать целая армия консультантов, стоящая огромных денег. К этой категории часто относятся системы CRM и CMS. Возникает подозрение, что само по себе это программное обеспечение не может *делать* ничего, оно лишь служит предлогом привести к вам эту армию консультантов, выставляющих счета в размере 300 долларов за час работы. Консалтинговое ПО маскируется под коробочное, но высокая цена его реализации делает его более близким к *заказному* ПО.

## *2. Заказное программное обеспечение*

Внутрифирменное ПО предназначено для работы только в конкретных условиях одной компании. Это значительно облегчает его разработку. Можно довольно точно оценить среду, в которой оно будет выполняться. Можно потребовать установить конкретную версию Internet Explorer, или Microsoft Office, или Windows NT 4.0 с *шестым* сервис-паком. Если вам нужна диаграмма, пусть ее строит Excel: у каждого в отделе установлен Ex-

cel. (Но попробуйте так поступить в коробочном продукте, и вы лишитесь половины потенциальных клиентов.)

Корпоративные разработчики не любят признаваться, что в программном обеспечении, разрабатываемом собственными силами, юзабилити уделяется небольшое внимание, поскольку работать с этим ПО будут люди, у которых нет другого выбора, им приходится работать с тем, что дадут, да и количество их невелико. И ошибки в таком ПО обычно встречаются чаще, чем в коробочном. Здесь важнее скорость разработки. Поскольку стоимость разработки ложится на бюджет лишь одной компании, объем ресурсов, которые можно привлечь для разработки, оказывается значительно меньше. Это Microsoft может позволить себе потратить $200 000 000 на разработку операционной системы, которая обычному человеку обойдется всего в $98. Но когда Чаттануга Рэйлроуд разрабатывает внутрифирменную трейдинговую платформу чучу, надо проверить, оправданны ли такие расходы для одной компании. Чтобы получить приемлемую прибыль от инвестированного капитала, разработчики внутрифирменного ПО не могут позволить себе тратить столько, сколько можно было бы потратить на разработку коробочного продукта. Фактически одно из главных отличий внутрифирменного ПО от коробочного состоит в том, что начиная с некоторого момента дополнительные усилия, направленные на повышение его надежности или простоты эксплуатации, становятся резко убыточными; для коробочного же продукта последний 1 процент, потраченный на повышение стабильности и простоты использования, может обеспечить решающее преимущество в конкуренции. Поэтому, как ни печально, большинство внутрифирменного ПО весьма плохо делает даже то, что должно было бы делать идеально. Это может угнетающе воздействовать на молодых и полных энтузиазма разработчиков, которых заставляют прекратить работу над ПО, поскольку оно уже «достаточно хорошее».

## *3. Встроенное ПО*

Уникальность этого ПО в том, что оно является частью какой-либо аппаратуры и, как правило, не обновляется. (Даже если это *технически* возможно. Уверяю вас, что никто не загружает обновление флэш-памяти своей микроволновой печи). Это совершенно особый мир. Требования к качеству очень высоки, потому что исправить ошибку уже нельзя. Программы выполняются процессором, который обычно гораздо слабее, чем в обычном настольном компьютере, поэтому масса времени тратится на

оптимизацию и ручную настройку. Скорость выполнения кода существеннее, чем его элегантность. Доступность устройств ввода-вывода ограничена. В системе GPS машины, которую я арендовал на прошлой неделе, были такие жалкие устройства ввода-вывода, что юзабилити ее производила тяжелое впечатление. Никогда не пробовали ввести адрес в устройство без клавиатуры? Или искать дорогу по карте размером с часы Casio? Но я отклонился от темы.

## *4. Игры*

Игры уникальны в двух отношениях. Во-первых, экономика разработки игр ориентирована на создание хитов. Некоторые игры становятся хитами, но гораздо большее их количество проваливается, и если вы хотите зарабатывать деньги на производстве игр, то должны это учитывать и создать портфель игр в расчете, что один блокбастер компенсирует убытки всех остальных. Это больше похоже на создание кинокартин, чем программ.

Более серьезная проблема разработки игр в том, что у них только одна версия. Сыграв до конца в *Duke Nukem 3D* и убив большого босса, никто не собирается делать обновление до *Duke Nukem 3.1D*, в котором будут исправлены некоторые ошибки и добавлено новое оружие. Это скучно. Поэтому в играх существуют такие же требования к качеству, как во встроенном ПО, и крайняя финансовая необходимость добиться успеха с первого раза, тогда как у разработчиков коробочного ПО, возящихся со своими DLL, есть хотя бы надежда, что если версия 1.0 дрянь и ее никто не берет, то 2.0 будет лучше, и ее станут покупать.

## *5. Разовые программы*

Пятая область, упоминаемая для полноты, это одноразовый код. Это временный код, который пишется, чтобы получить что-то еще, и становится ненужным, как только оно получено. Например, вам может понадобиться небольшой сценарий оболочки, чтобы перевести полученный файл входных данных в формат, который необходим для каких-то других целей, и такая операция оказывается разовой. В одноразовом коде не таится никаких опасностей, хотя существует закон, согласно которому разовый код может внезапно стать внутрифирменным и какой-нибудь обнаруживший его бизнес-администратор решит, что на этом можно развернуть

целый бизнес, – и *hate*! Появляется еще одна консалтинговая компания, предлагающая непрочные «решения масштаба предприятия».

## *Свой мир надо знать*

Мы живем в разных мирах. Ну и что? Какое это может иметь значение? А вот какое. Когда вы читаете некую книгу о программных методологиях, написанную профессиональным гуру/консультантом в области разработки программного обеспечения, то скорее всего речь в ней идет о внутрифирменной разработке корпоративного программного обеспечения. Не коробочных продуктов, не встроенного ПО и уж явно не игр. Почему? Потому что нанимают этих гуру корпорации. Они платят деньги. (Поверьте мне, id Software не станет приглашать Эда Йордона, чтобы он рассуждал о структурном анализе.)

В последнее время я внимательно присматривался к экстремальному программированию (XP). Для многих типов проектов XP и другие ускоренные методы оказываются глотком свежего воздуха по сравнению с неестественными, скучными и порочными «процессами», применяемыми не в одной из софтверных компаний, и не вызывает сомнения, что такие технологии, как разработка на основе тестирования, просто потрясают, если вы сумеете их применить. Но в этом и загвоздка: *если вы сумеете их применить*. В XP есть целый ряд подводных камней, которые могут стать опасными в конкретной ситуации. Например, никто не добился успеха в применении разработки на основе тестирования для создания GUI. Я построил целый сервер веб-приложения с помощью разработки на основе тестирования, и он работал потрясающе, потому что сервер приложения занят в основном обработкой строк. Но я не смог применить *ни один вид* автоматизированного тестирования для создания моих GUI. В лучшем случае можно воспользоваться автоматизированным тестированием, которое не поможет, если надо реализовать буксировку методом drag and drop.

XP также предполагает простоту выполнения рефакторинга, особенно при наличии тестов, подтверждающих его корректность. В монолитных проектах, особенно разрабатываемых внутри фирм и независимо от внешних факторов, так оно и есть. Но я *не могу* произвольно изменить схему продукта FogBUGZ, производимого моей компанией, потому что многие наши клиенты написали для него свой код, а если я сделаю его неработоспособным, они не станут покупать следующую версию, а тогда я не смогу заплатить программистам, и весь этот карточный домик рухнет. Тщатель-

ным анализом и проектированием нельзя пренебрегать, если вы собираетесь корректно взаимодействовать с другими. Вот я и пытаюсь выяснить, как применять лучшие из ускоренных методологий в разработке коробочного ПО и определять, когда это невозможно.

Как правило, законы разработки ПО одинаковы для любых типов проектов, но не всегда. Когда кто-нибудь рассказывает о методологии, надо подумать о том, применима ли она в *вашей конкретной* работе. Учитывайте то, в какой области работает этот человек. Все мы можем чему-то поучиться друг у друга. В *каждой сфере есть свои источники трудностей*. Если вы корпоративный разработчик приложений для сервера Java, то не стоит вводить людей в заблуждение и объяснять причину отсутствия у вас «кошмара DLL» тем, что Java сам по себе более совершенный язык программирования. Вы живете без кошмара DLL, *потому что ваша программа работает только на одной машине*. Если вы разрабатываете игры, перестаньте жаловаться на то, как медленно работает интерпретируемый байт-код. *Он не для вас*. Он для внутрифирменных разработчиков, программу которых пытаются «завалить» четыре оператора, ведущих бухгалтерию. Уважайте и понимайте проблемы теста/хлебных крошек, которые есть у каждого, и давайте сделаем Usenet более цивилизованной средой обитания!

ГЛАВА ТРИНАДЦАТАЯ

# Создание прототипов на бумаге

16 мая 2003 года, пятница

Много-много лет назад бригада разработчиков Excel пыталась выяснить, стоит ли давать пользователям возможность перетаскивать ячейки с помощью мыши. Паре стажеров было поручено «слепить прототип» для проверки юзабилити с помощью такого новейшего продукта, как Visual Basic 1.0. Создание прототипа растянулось на целое лето, потому что он должен был копировать значительную часть действительной функциональности Excel, иначе нельзя было бы провести действительное испытание юзабилити.

Вывод из тестирования юзабилити? Да, это хорошая функция! Ответственный за нее программист потратил *чуть ли* не неделю и реализовал эту функцию перетаскивания. Юмор, разумеется, в том, что весь *смысл* прототипирования состоит в том, чтобы «сэкономить время».

Годом позже еще одна засекреченная команда Microsoft построила полный прототип нового интерфейса пользователя с помощью новейшего продукта Asymetrix Toolbook[1] (с трудом верится, но эта штука жива до сих пор). Создание прототипа заняло около года. Что это получился за продукт? Microsoft Bob,[2] эквивалент PCjr в мире программного обеспечения. Еще один напрасно сделанный прототип.

Я в принципе отказался от создания прототипов программ. Если прототип может делать то же, что и продукт, он и сам может *быть* продуктом, а если не может, то от него мало толку. К счастью, существует более удач-

---

1 См. *www.asymetrix.com/en/toolbook/index.asp*.

2 См. *toastytech.com/guis/bob2.html*.

ная идея: прототип на бумаге, который отлично решает эту проблему и проблему айсберга[1] одним ударом.

Бумажный прототип штука несложная – это всего лишь лист бумаги, на котором нарисован (хотя бы и карандашом) макет интерфейса пользователя. Чем неаккуратнее, тем лучше. Покажите его нескольким людям и спросите, как бы они выполнили такое задание. Вы не рискуете завязнуть в обсуждении цветов и шрифтов, потому что цветов и шрифтов нет – одни карандашные наброски. А поскольку ясно, что вы не слишком много сил потратили, никто не боится высказывать свое мнение из опасения задеть ваши чувства.

Создавать прототипы на бумаге намного дешевле, чем с помощью программных средств, и при этом можно проводить настоящие тесты юзабилити; просто вы стоите рядом – с резинкой, карандашом и ножницами. Когда проверяющий сообщает вам, что он сделал («щелкнул здесь!»), вы просто тут же все переставляете. Проверка помощника? Делаете лист бумаги для каждой страницы помощника и готовите какие-нибудь карточки с возможными сообщениями об ошибке.

Подробнее о прототипировании на бумаге рассказано в книге Кэролин Снайдер.[2] Это хороший справочник для всех разработчиков пользовательских интерфейсов, пригодный и для новичков.

---

[1] См. главу 25.

[2] Snyder, Carolyn (Кэролин Снайдер) «Paper Prototyping: The Fast and Easy Way to Design and Refine User Interfaces» (Макетирование на бумаге: быстрый и простой способ проектирования и уточнения интерфейса пользователя), Morgan Kaufmann, 2003.

ГЛАВА ЧЕТЫРНАДЦАТАЯ

# Не дайте астронавтам от архитектуры запугать себя

21 апреля 2001 года, суббота

Великие умы, размышляя о проблемах, обнаруживают закономерности. Они задумываются о пересылке файлов текстового процессора, а потом о пересылке электронных таблиц, и открывают общую закономерность: люди посылают файлы. Так возникает один уровень абстракции. Затем они поднимаются выше еще на один уровень: люди *посылают* файлы, но веб-броузеры тоже «*посылают*» запросы для получения веб-страниц. А если вдуматься, то вызов метода объекта совершенно аналогичен посылке сообщения объекту! Та же самая закономерность! Все это операции *посылки*, и наш мыслитель изобретает новую, более высокую и широкую абстракцию, названную им *посылкой сообщений* (*messaging*), так что теперь все становится *совершенно* неясным, и никто уже не понимает, о чем они говорят.

Забираясь в подобном абстрагировании слишком высоко, вы лишаете себя кислорода. Временами эти мудрые мыслители не могут вовремя остановиться и создают нелепые универсальные абстрактные картины вселенной, которые замечательны, но не имеют абсолютно никакого смысла.

Такого рода людей я называю архитектурными астронавтами. Очень трудно посадить их писать код или проектировать программы, потому что они непрерывно размышляют об архитектуре. Это настоящие астронавты, потому что они настолько выше содержащей кислород атмосферы, что непонятно, *чем* они дышат. Они стремятся работать в очень крупных компаниях, которые могут себе позволить содержать массу ничего не производящих людей с высокими званиями, ничего не вносящих в конечные результаты.

Иллюстрацией может служить следующий довольно свежий пример. Типичный архитектурный астронавт, рассматривая утверждение о том, что «Napster – это одноранговая сеть для загрузки музыки», проигнорирует все, кроме указания на архитектуру, его заинтересует только одноранговость Napster, а возможность ввести название песни и *сразу* начать ее слушать оставит его безразличным.

Теперь у них только и разговоров: одноранговое одно, другое, третье. Внезапно появляются одноранговые конференции, одноранговые фонды венчурных капиталов и даже одноранговое обратное движение, когда слабоумные бизнес-журналисты радостно пускают слюни, копируя друг у друга истории: «Конец одноранговых сетей!»

Архитектурные журналисты задают вопросы типа: «Нельзя ли представить себе программу типа Napster для загрузки *чего угодно*, а не только песен?» Потом они строят приложения типа Groove, которые, по их мнению, обладают *более* общими возможностями, чем Napster, но при этом пренебрегают некоей незаметной функцией, позволяющей ввести название песни, а потом ее слушать, – той функцией, которая и была нам нужна в первую очередь. Речь о том, что не понят весь смысл. Если бы Napster *не был* одноранговой сетью, но *позволял* ввести название песни и слушать ее, его популярность от этого не уменьшилась бы.

Еще архитектурные астронавты очень любят изобрести какую-нибудь новую архитектуру и утверждать, что она решает некую проблему. Java, XML, Soap, XML-RPC, HailStorm, .NET, Jini – нет сил продолжать. И это только в течение одного года!

Я не хочу сказать, что эти архитектуры плохи... ни в коем случае. Это вполне добротные архитектуры. Что меня раздражает, так это неимоверная рекламная шумиха, которой они сопровождаются. Помните информационный доклад Microsoft по поводу .NET?[1]

> Представляя собой новое поколение настольных платформ Windows, Windows .NET поддерживает высокую эффективность, творческую активность, менеджмент, отдых и многое другое, будучи нацеленной на установление контроля пользователей за их цифровой жизнью.

[1] «Microsoft .NET: Realizing the Next Generation Internet» (Microsoft .NET: Интернет следующего поколения), Microsoft, June 22, 2000. На веб-сайте Microsoft этого документа больше нет, но его копия есть на *web.archive.org/web/20001027183304* и *http://www.microsoft.com/business/vision/netwhitepaper.asp*.

Это было девять месяцев назад. В прошлом месяце нам объявили о Microsoft HailStorm. Соответствующее информационное сообщение[1] гласит:

> Люди не владеют окружающей их технологией... HailStorm служит тому, чтобы технологии в вашей жизни действовали совместно для вашего блага и под вашим контролем.

Отлично, теперь высокотехнологичная лампа у меня в квартире перестанет мерцать.

Microsoft не одинока. Вот цитата из информационного сообщения Sun Jini:[2]

> Данные три обстоятельства (вы – новый системный администратор, нигде вокруг нет компьютеров, и один компьютер находится повсюду) должны в совокупности повлиять в лучшую сторону на использование компьютеров в качестве компьютеров – путем размывания границ отдельных компьютеров, повсеместного проникновения одного компьютера и превращения работы с компьютером в такую же простую вещь, как воспроизведение DVD в домашнем театре.

И *не напоминайте* мне об удобрении, которое Джордж Гилдер внес в распространение Java:[3]

> Фундаментальный и исторический прорыв в технологии...

Вот явные свидетельства того, что вы подверглись атаке архитектурного астронавта: невероятная напыщенность, эпическая высокопарность,

---

1 «An Introduction to Microsoft HailStorm» (Введение в Microsoft HailStorm), Microsoft, March 2001. На веб-сайте Microsoft этого документа больше нет, но его копия есть на *web.archive.org/web/20010413105836* и *http://www.microsoft.com/net/hailstorm.asp*.

2 «Why Jini Technology Now?» (Почему технология Jini появилась сейчас?), Sun.com, 1999. См. *www.sun.com/jini/whitepapers/whyjininow.html*.

3 «The Coming Software Shift» (Грядущие перемены в ПО), George Gilder, *Forbes ASAP*, August 28, 1995.

хвастовство, полный отрыв от реальности. И людям это нравится! Бизнес-пресса с ума сходит!

По каким непостижимым причинам на многих производит такое огромное впечатление какая-то скучная архитектура, часто сводящаяся лишь к новому формату RPC или новой виртуальной машине? Вполне возможно, что все это хорошие архитектуры, и несомненно, что разработчики выиграют от их использования, но они *не* являются, я подчеркиваю, *не являются* достойной заменой мессии, въезжающему в Иерусалим на белом осле, или наступлению всеобщего мира. Нет, Microsoft, компьютеры *не* начнут внезапно угадывать наши мысли и автоматически делать то, что нам нужно, лишь благодаря тому, что все заведут учетную запись в системе Passport. Нет, Sun, мы *не* сможем анализировать результаты данных корпоративных продаж с такой же легкостью, с какой «просматриваем диск DVD в домашнем кинотеатре».

Помните, что эти архитекторы решают те проблемы, которые, как им кажется, они могут решить, а не те проблемы, которые было бы *полезно* решить. Возможно, что SOAP + WSDL – это самая горячая новость, но с ее помощью вы не сделаете ничего такого, чего не могли делать раньше с помощью других технологий – если вам это действительно было нужно. Вся эта нирвана распределенных сервисов, о которой треплются архитектурные астронавты, уже была обещана нам ранее – посредством DCOM, JavaBeans, OSF DCE или CORBA.

Прекрасно, что теперь мы можем передавать сообщения в формате XML. Ура! Меня это должно волновать не более чем то, что товары в мой супермаркет привозят со склада грузовики. Скучно. Лучше расскажите мне, что я теперь смогу сделать нового, чего не мог сделать раньше, о, Астронавты, или оставайтесь там в своем космосе и не отнимайте у меня мое время.

ГЛАВА ПЯТНАДЦАТАЯ

# Огонь и движение

6 января 2002 года, воскресенье

Иногда у меня ничего не получается.

Конечно, я прихожу в офис, бесцельно слоняюсь, каждые 10 секунд проверяю почту, брожу по Интернету и даже делаю что-нибудь, не требующее умственных усилий, например оплачиваю счета American Express. Но втянуться в писание кода как-то не получается.

Обычно такие приступы непродуктивности длятся один или два дня. Но были в моей карьере программиста и такие случаи, когда я неделями не был способен ничего сделать. Как говорят, нет вдохновения. Не попасть в ритм. Никуда не попасть.

У всех бывают перемены настроения; у одних они мягкие, у других – более выраженные и даже разрушительные. И периоды непродуктивности коррелируют с периодами плохого настроения.

Я вспоминаю при этом, что некоторые исследователи считают, что по существу люди *не могут* контролировать, чем они питаются, поэтому любая попытка диеты неизбежно становится краткосрочной и человек возвращается к своему естественному весу. Возможно, как разработчик программного обеспечения я действительно не в состоянии управлять своей продуктивностью и приходится смириться с наличием более продуктивных и менее продуктивных периодов в надежде, что в среднем я сумею писать достаточное количество строк кода, чтобы меня не выгнали с работы.

Но я заметил, что начиная с самой первой моей работы я как разработчик в среднем лишь два или три часа могу продуктивно писать код, и это

меня изумляет. Когда я летом стажировался в Microsoft, то мой коллега-стажер рассказал мне, что фактически он ежедневно активно работал с 12 до 5. Пять часов, минус обед, и его бригада *восхищалась* им, потому что ему все же удавалось сделать гораздо больше среднего. Я обнаружил, что со мной происходит то же самое. Я испытываю легкое чувство вины, замечая, как напряженно, по-видимому, работают все остальные, а у меня получается два или три часа настоящей работы в течение дня, и тем не менее я всегда был одним из самых результативных членов бригады. Видимо по этой причине «Peopleware»[1] и экстремальное программирование настаивают на отмене сверхурочной работы и на строго 40-часовой рабочей неделе, будучи уверенными в том, что объем продукции команды от этого не уменьшится.

Но тревожат меня не те дни, когда я могу сделать «всего» двухчасовую работу. Меня тревожат дни, когда я не могу сделать *ничего*.

Я много размышлял над этим. Я пытался вспомнить, в какой момент моей карьеры я делал больше всего работы. Вероятно, это было, когда Microsoft поместила меня в шикарный новый офис с громадными окнами, выходившими в прелестный дворик с множеством цветущих вишен. Все у меня ладилось. Я месяцами работал безостановочно, выдавая детальную спецификацию Excel Basic – монументальную гору бумаги, где со всеми мыслимыми подробностями описывались гигантская объектная модель и среда программирования. Я буквально не мог остановиться. Когда мне пришлось поехать в Бостон на MacWorld, я взял с собой лэптоп и описывал класс Window, сидя на чудесном балконе в HBS.

Когда попадаешь в струю, нетрудно двигаться в ней. Многие мои дни проходят так:

1. Прихожу на работу.
2. Проверяю почту, хожу по Интернету и т. д.
3. Решаю, что можно пообедать, прежде чем браться за работу.
4. Возвращаюсь с обеда.
5. Проверяю почту, хожу по Интернету и т. д.
6. Наконец, решаю, что пора начать работать.
7. Проверяю почту, хожу по Интернету и т. д.
8. Снова решаю, что теперь *действительно* пора начать работать.

---

[1] Том Демарко и Тимоти Листер «Человеческий фактор: успешные проекты и команды», 2-е издание, СПб.: Символ-Плюс, 2005.

9. Запускаю чертов редактор.
10. Без передышки пишу код, пока не обнаруживаю, что уже полвосьмого вечера.

По-видимому, между этапами 8 и 9 есть какой-то дефект, потому что мне не всегда удается преодолеть находящуюся между ними пропасть. *Самое* трудное для меня – это начать. Тело, находящееся в покое, стремится сохранять состояние покоя. В моем мозгу есть что-то невероятно тяжелое, из-за чего его крайне трудно разогнать, но когда он разгонится до полной скорости, поддерживать ее нетрудно. Как велосипед, оборудованный для автономной поездки по пересеченной местности:[1] когда впервые пытаешься тронуть с места велосипед со всем тем, что на него навьючено, то уходит много сил, но когда он покатился, ехать становится так же легко, как на велосипеде без дополнительных принадлежностей.

Может быть, в этом и лежит ключ к продуктивности: *надо просто начать*. Может быть, если программирование парами эффективно, то это потому, что когда вы планируете со своим приятелем сеанс парного программирования, вы подстегиваете друг друга начать работу.

Когда я был в Израиле парашютистом, один генерал произнес перед нами небольшую речь по поводу стратегии. Когда пехота ведет бой, сообщил он нам, у нее может быть только одна стратегия: огонь и движение. Вы движетесь в сторону противника и одновременно ведете огонь из своего оружия. Ваш огонь вынуждает противника спрятать голову, поэтому он не может вести ответный огонь. (Это и имеет в виду солдат, когда кричит «прикрой меня». Это значит «стреляй в противника, чтобы он опустил голову и не мог стрелять в меня, пока я буду перебегать улицу». И это действует.) Движение позволяет захватить территорию и приблизиться к врагу, и тогда ваши выстрелы вернее поразят цель. Если вы не движетесь, то развитие событий станет определять противник, а это нехорошо. Если вы не ведете огонь, противник станет вести огонь по вам и уложит вас.

Я надолго запомнил это. Я заметил, что почти каждый тип военной стратегии, от воздушного боя до больших морских маневров, основан на идее «огня и движения». Мне понадобилось еще 15 лет, чтобы понять, что по принципу «огня и движения» вы достигаете целей в жизни. Надо хоть немного продвигаться вперед, каждый день. Пусть ваш код кривой, с ошибками и никому не нужен. Если вы неуклонно движетесь вперед, совершенствуя код и исправляя ошибки, время на вашей стороне. Будьте ос-

[1] См. *joelspolsky.com/biketrip/*.

торожны, когда конкуренты открывают по вам огонь. Не хотят ли они просто заставить вас отвечать на их очереди, чтобы вы не смогли двигаться дальше?

Посмотрите на историю стратегий доступа к данным, вышедших из недр Microsoft. ODBC, RDO, DAO, ADO, OLEDB, а теперь ADO.NET – все новые! Что это – технологическая необходимость? Результат некомпетентности проектировщиков, вынужденных каждый год изобретать новый способ доступа к данным? (Весьма возможно, что именно так.) Но конечный результат – это огонь прикрытия. У конкурентов не остается иного выбора, кроме как тратить все свое время на то, чтобы портировать свои продукты и поспевать за новыми технологиями, – то время, когда они могли бы писать свои новые функции. Вглядитесь в панораму разработки ПО. Хороших результатов добиваются те компании, которые в наименьшей степени зависят от больших компаний и не вынуждены тратить каждый свой бизнес-цикл на то, чтобы догонять, заново реализовывать и исправлять ошибки, появляющиеся только в Windows XP. А спотыкаются те компании, которые слишком много времени потратили, пытаясь по кофейной гуще угадать, в каком направлении двинется Microsoft дальше. Люди озабочены появлением .NET и решают переделать под .NET всю свою архитектуру, потому что считают себя вынужденными это сделать. Microsoft ведет по вам огонь, и это всего лишь огонь прикрытия, чтобы обеспечить движение вперед для себя и не дать двигаться вперед вам, потому что, малыш, в этом и состоит игра. Вы собираетесь вводить поддержку HailStorm?[1] SOAP?[2] RDF?[3] Вы поддерживаете их потому, что это нужно вашим клиентам, или потому, что кто-то открыл по вам огонь и вы чувствуете себя обязанным ответить на него? Отделы продаж крупных компаний хорошо понимают, что такое огонь прикрытия. Они любезно говорят своим клиентам: «Конечно, вы не обязаны покупать именно наш продукт. Купите у того, кто покажется вам лучше. Только убедитесь, что предлагаемый вам продукт поддерживает XML / SOAP / CDE / J2EE, потому что иначе вы связаны по рукам и ногам».[4] И когда небольшие компании пытаются выйти на рынок соответствующего продукта, только и слышно, как руководители технических

---

1 См. *wmf.edittbispage.com/discuss/msgReader$3194?mode=topic*.

2 См. *radiodiscuss.userland.com/soap*.

3 См. *www.w3.org/RDF/*.

4 Winer, Dave (Дейв Винер) «The Micro Channel Architecture» (Микроканальная архитектура), DaveNet, July 6, 2001. См. *davenet.userland.com/2001/07/ 06/theMicroChannelArchitecture*.

отделов послушно твердят, как попугаи: «А у вас есть J2EE?» И им приходится убивать все свое время на встраивание J2EE, даже если в этом нет особого смысла, и терять возможность действительно выделиться. Это функция «для галочки»: вы реализуете ее для того, чтобы поставить галочку в списке характеристик, но она никому не нужна, и никто не будет ей пользоваться. Это огонь прикрытия.

Для небольших компаний, таких как моя, «огонь и движение» означает две вещи. Вы должны привлечь время на свою сторону,[1] вы должны ежедневно продвигаться вперед. Рано или поздно вы победите. Все, что мне удалось вчера сделать, – это немного улучшить цветовую схему FogBUGZ. Это нормально. Все постепенно делается лучше. С каждым днем наша программа становится все лучше, и покупателей становится все больше, и это самое главное. Пока мы не достигли размеров Oracle, нам не нужно выдумывать грандиозные стратегии. Нам надо просто приходить каждое утро и пытаться запустить редактор.

[1] См. главу 36.

ГЛАВА ШЕСТНАДЦАТАЯ

# Мастерство

1 ДЕКАБРЯ 2003 ГОДА, ПОНЕДЕЛЬНИК

Создание программного обеспечения – это не промышленный процесс. В 1980-х годах все были в ужасе оттого, что японские софтверные компании начали создавать «фабрики программного обеспечения», которые должны были выдавать высококачественный код конвейерным способом. В этом не было смысла тогда, нет его и сейчас. Затолкав кучу программистов в помещение и усадив их ровными рядами, вы не добьетесь снижения количества ошибок.

Если написание кода – не конвейерное производство , то что это? Некоторые предложили окрестить его *мастерством*. Это тоже не совсем верно, и что бы вы мне ни говорили, я считаю, что окно диалога в Windows, в котором у вас спрашивают, как вы хотите проиндексировать файл подсказки, никоим образом не может ассоциироваться у культурного человека со словом «мастерство».

Написание кода – не *производство*. Оно не всегда оказывается мастерством (но бывает). Это *дизайн*. *New York Times Magazine* восторгается iPod[1] и тем, что Apple – одна из немногих компаний, которые умеют превратить хороший дизайн в источник новой стоимости.

Но хватит о дизайне. Поговорим немного о мастерстве – о том, что это такое и как его распознать.

---

[1] Walker, Rob (Роб Уокер) «The Guts of a New Machine» (Подноготная новой машины), *The New York Times*, November 30, 2003, late edition – final, section 6, page 78, column 1.


*Любезно предоставлено Apple Computer, Inc. Дополнительная информация на: http://www.apple.com/pr/photos/ipod/03ipod.html*

Я бы хотел рассказать о фрагменте кода, который я переписываю для CityDesk 3.0: код импорта файлов. (Реклама: CityDesk – простая в обращении система управления контентом, выпускаемая моей компанией.)

Спецификация выглядит настолько просто, насколько это возможно для какого-либо фрагмента кода. Пользователь выбирает файл в стандартном диалоговом окне, а программа копирует этот файл в базу данных CityDesk.

Она послужила отличным примером того случая, когда «последний 1 процент кода отнимает 90 процентов общего времени». Первый набросок кода выглядел так:

1. Открыть файл.
2. Прочесть его целиком в один большой массив байтов.
3. Запомнить массив байтов в записи.

Работало отлично. Для файлов разумных размеров – практически мгновенно. Было несколько мелких ошибок, которые я постепенно исправил.

Большая ошибка всплыла, когда я провел испытание под нагрузкой, перетащив в CityDesk файл размером 120 Мбайт. Да, едва ли часто встречаются случаи отправки веб-сайту 120-мегабайтных файлов. По-видимому, весьма редко. Но и исключить этого тоже нельзя. Код сделал требуемое, но это заняло почти минуту, и никакой визуальной связи с пользователем при

этом не было – казалось, что приложение замерло и полностью заблокировалось. Очевидно, это не идеальное поведение.

С точки зрения интерфейса пользователя для таких долгих операций следовало бы показывать какой-нибудь индикатор состояния вместе с кнопкой Cancel. В идеале должна была бы сохраниться возможность выполнять в CityDesk другие действия, а копирование файла продолжить в фоновом режиме. Очевидно, это можно было сделать тремя способами:

1. В одном потоке, часто опрашивая события ввода.
2. Запустив второй поток и синхронизировав его с большой осторожностью.
3. Запустив второй процесс и синхронизировав его с меньшей осторожностью.

Мой опыт работы по варианту #1 подсказывает, что он никогда не работает хорошо. Слишком трудно обеспечить безопасное выполнение любого кода вашего приложения во время осуществления операции копирования файла. А Эрик Рэймонд убедил меня, что обычно потоки оказываются худшим решением, чем отдельные процессы;[1] действительно, годы практики показали мне, что программирование параллельных потоков значительно увеличивает степень сложности и приводит к появлению целых новых классов опаснейших гейзенбагов.[2] Вариант #3 показался мне хорошим решением, тем более что наша реальная база данных была многопользовательской и нисколько не возражала против одновременного нападения на нее со стороны кучи процессов. Этим я и собираюсь заняться после каникул по случаю Дня Благодарения.

Обратите, однако, внимание, что получилось. Мы перешли от чтения файла и записи его в базу данных к существенно более сложной задаче: запустить дочерний процесс, дать *ему* указание прочесть файл и записать его в базу данных, добавить индикатор выполнения операции и кнопку отмены в дочерний процесс и создать какой-то механизм для того, чтобы дочерний процесс сообщил родительскому о том, что файл получен и его можно показать. Еще надо потрудиться над передачей дочернему процессу аргументов командной строки и разумным поведением фокуса окна, а также обработать случай выключения пользователем машины в процессе копирования файла. Навскидку, мне понадобится в десять раз больше

---

1 См. *www.faqs.org/docs/artu/ch07s03.html#id2923889*.

2 Подробнее о гейзенбагах см. *c2.com/cgi/like?HeisenBug* (или *http://ru.wikipedia.org/wiki/Гейзенбаг*).

кода, чтобы корректно обращаться с большими файлами, и лишь 1 процент наших пользователей, возможно, столкнется с этим кодом.

Конечно, есть программисты, которые заявят, что моя новая архитектура с дочерними процессами хуже первоначальной. Она «раздута» (потому что появилось много новых строк кода). Вероятность ошибок в ней больше, потому что появилось много новых строк кода. Она не вызывается необходимостью. Она даже служит символом неполноценности Windows как операционной системы, сказали бы они. «Зачем вся эта возня с индикаторами? – хмыкнут они. – Нажимаешь Ctrl+Z, а затем, когда надо, выполняешь ls –l и убеждаешься, что файл растет!»

Мораль этого рассказа в том, что иногда исправление 1 процента неисправностей требует 500 процентов усилий. Это не уникальная особенность программного обеспечения, отнюдь; руководя всеми этими строительными проектами, я могу сказать точно. На прошлой неделе наш подрядчик наконец-то положил завершающие мазки в новых офисах Fog Creek.[1] Они состояли в установке на входных дверях блестящих голубых акриловых панелей в алюминиевой окантовке, крепящейся шурупами через каждые 20 см. Внимательно вглядевшись в картинку, можно обнаружить, что алюминиевая окантовка окружает каждую дверь. Там, где смыкаются две створки, два вертикальных куска окантовки располагаются рядом друг с другом. По картинке судить трудно, но в действительности шурупы средних полосок *почти*, но *не точно*, выровнены. Расхождение миллиметра 2, не больше. Делавший двери мастер вымерил все точно, но он устанавливал окантовку, пока двери лежали на земле, а не висели на своем месте, и когда их повесили – увы, стало заметно, что шурупы выровнены не точно.

Это не столь уж редкий случай: у нас в офисе множество шурупов, которые не идеально выровнены. Проблема в том, что исправлять это после того, как все дырки уже просверлены, очень дорого. Так как правильное положение шурупов находится всего в двух миллиметрах от нынешнего, нельзя взять и просто просверлить в дверях новые отверстия; возможно, придется заменять всю дверь. Результат того не стоит. Это еще один пример, показывающий, как исправление 1 процента неисправностей требует 500 процентов усилий, и объясняющий, почему в нашем мире так много вещей, которые хороши на 99 процентов, а не на все 100. Наш архитектор восторгался неким действительно очень дорогим зданием в Аризоне, в котором выровнены все шурупы.

---

[1] См. *www.joelonsoftware.com/articles/BionicOffice.html.*


То же относится к такому атрибуту программного обеспечения, который большинство людей называют *мастерством*. Когда программа сделана подлинным мастером, все шурупы выровнены. Когда вы выполняете какую-нибудь редкую операцию, приложение ведет себя разумно. На то, чтобы редкие операции выполнялись абсолютно точно, ушло больше усилий, чем на разработку основного кода. Возможно, потребовалось 500 процентов усилий, чтобы обработать 1 процент случаев.

Мастерство, конечно, обходится крайне дорого. Позволить себе это можно только при разработке программ для массового покупателя. Увы, приложения для учета кадров, разработанные внутри страховых компаний для своих нужд, никогда не достигнут такого уровня мастерства просто потому, что у них *нет достаточного количества пользователей*, среди которых можно было бы распределить дополнительные издержки. В компании же, производящей коробочный продукт, именно такой уровень мастерства приводит в восхищение пользователей и обеспечивает долгосрочные преимущества в конкурентной борьбе, поэтому я потрачу время и сделаю все, как надо. Потерпите.

ГЛАВА СЕМНАДЦАТАЯ

# Три ложных постулата информатики

22 августа 2000 года, вторник

Не хочу никого обидеть, но есть три важных положения информатики, которые, честно говоря, неправильны, и люди начинают это замечать. Можете не обращать на них внимания, но на свой страх и риск.

Я уверен, что их больше, но эти три настолько заметны, что я прихожу в отчаяние:

1. Трудность поиска обусловлена необходимостью найти достаточное количество результатов.
2. Текст без сглаживания шрифтов (anti-aliasing) выглядит лучше.
3. Сетевое ПО должно делать сетевые ресурсы неотличимыми от локальных.

Все, что я могу сказать по этому поводу:

1. Неверно.
2. Неверно.
3. НЕВЕРНО!

Сделаем короткий обзор.

## *Поиск*

В большинстве научных исследований по проблемам поиска прослеживается явная *одержимость* проблемами типа «что делать, если вы ищете “машину”, а в документе написано “автомобиль”?»

Действительно масса исследований посвящена таким понятиям, как *поиск по основе*, когда из искомого слова выделяется его несклоняемая часть,

и при поиске «поиска» вы находите документы со словами «искал» или «искомый».

Когда в Интернете появились большие поисковые механизмы типа Altavista, они похвалялись тем, что находят мириады результатов. Поиск в Altavista «Joel on Software» дает 1033555 страниц. Разумеется, это бессмысленно. В Интернете всего насчитывается, наверное, миллиард страниц. Сократив поиск с миллиарда до миллиона, Altavista не принесла мне абсолютно никакой пользы.

*Действительная* проблема поиска в том, как *сортировать* результаты. В оправдание ученых-компьютерщиков надо сказать, что никто даже *не замечал* этого, пока не стали индексировать такие гигантские собрания документов, как Интернет.

Но кое-кто заметил. Ларри Пэйдж и Сергей Брин в Google поняли, что упорядочить страницы в правильной последовательности важнее, чем схватить все мыслимые страницы. Их алгоритм PageRank[1] дает замечательный способ отсортировать мириады найденных результатов так, чтобы тот, который вам нужен, оказался в верхней десятке. И в самом деле, поищите «Joel on Software» в Google, и вы обнаружите, что нужный результат идет первым. В Altavista его не нашлось даже на первых пяти страницах, дальше которых я смотреть не стал.

## *Сглаживание текста*

Сглаживание (antialiasing) было придумано еще в 1972 году в МТИ, в Architecture Machine Group, вошедшей впоследствии в знаменитую Media Lab. Идея заключается в том, что если у вас цветной дисплей с низким разрешением, то с помощью оттенков серого цвета можно создать иллюзию высокого разрешения. Вот как это выглядит на деле:

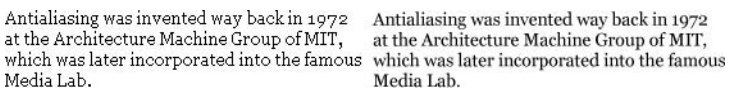
Обратите внимание на то, что обычный текст слева – четкий и резкий, а сглаженный текст справа – размыт по краям. Если глянуть искоса или не-

[1] См. *www.google.com/technology/index.html*.

сколько отодвинуться, то у обычного текста окажутся причудливые «ступеньки» из-за ограниченного разрешения экрана компьютера. А вот сглаженный текст будет выглядеть лучше и приятнее.

По этой причине сглаживание всем очень понравилось. Оно теперь есть всюду. В Microsoft Windows есть даже флажок, который включает сглаживание для любых текстов в системе.

В чем проблема? Если вы попробуете прочесть абзац сглаженного текста, он покажется размытым. Тут ничего не поделаешь, это так. Сравните следующие два абзаца:




Абзац слева не подвергнут сглаживанию. Абзац справа был сглажен с помощью Corel PHOTO-PAINT. Честно говоря, сглаженный текст смотрится *плохо*.

В конце концов, на это обратили внимание – в группе Microsoft Typography.[1] Там создали несколько прекрасных шрифтов типа Georgia и Verdana, которые «предназначены для облегчения чтения с экрана». По существу, вместо создания шрифта с высоким разрешением и последующих попыток уложить его в сетку пикселов они приняли сетку как «данность» и разработали шрифт, который точно в нее ложится. Кое-где *не обратили* на это внимание – в группе Microsoft Reader, где используется вид сглаживания, названный «ClearType», предназначенный для цветных LCD-экранов, и который, простите, все равно выглядит расплывчато, даже на цветном LCD-экране.[2]

Чтобы графики-профессионалы среди моих читателей не завалили меня разгневанными отзывами, я скажу, что сглаживание все равно остается замечательной технологией для двух вещей: заголовков и логотипов, в которых общий вид важнее возможности долго читать, и картинок. Сглаживание замечательно помогает уменьшать фотографические изображения.

---

1 Подробнее о Microsoft typography см. *www.microsoft.com/truetype/default.asp*.

2 «First ClearType screens posted», Microsoft.com, January 26, 2000. См. *www.microsoft.com/typography/links/News.asp?NID=1135*. С момента, когда я это написал, ClearType стал стандартной функцией в Microsoft Windows XP и значительно усовершенствовался.

## *Сетевая прозрачность*

С момента появления первых сетей Священным Граалем сетевого программирования стало создание программного интерфейса, позволяющего обращаться к удаленным ресурсам *точно так же*, как к локальным. Сеть становится «прозрачной».

Одним из примеров сетевой прозрачности служит знаменитый RPC (удаленный вызов процедур),[1] система, позволяющая вызывать процедуры (подпрограммы), выполняемые на других компьютерах в сети, точно так же, как если бы они выполнялись на локальном компьютере. Для достижения этого было потрачено неимоверное количество сил. Другой пример – построенная поверх RPC система Microsoft «распределенный COM» (DCOM),[2] позволяющая обращаться к объектам, выполняющимся на другом компьютере, как если бы они находились на данном компьютере.

Как будто логично, да?

Нет.

Есть три очень существенных различия между доступом к ресурсам на удаленной машине и доступом к ресурсам на локальной машине:

1. Доступность.
2. Латентность.
3. Надежность.

Когда вы обращаетесь к другой машине, весьма вероятно, что та машина недоступна или сеть недоступна. А скорость передачи данных по сети может обусловить длительное выполнение запроса, особенно если вы соединены через модем со скоростью 28.8 Кбит/с. Или на другой машине может возникнуть фатальный сбой, или сетевое соединение разорвется во время связи с другой машиной (кошка прыгнет через телефонный провод).

В любом надежном программном обеспечении, использующем сети, совершенно необходимо принимать во внимание эти обстоятельства. Применение программных интерфейсов, которые их скрывают, значительно повышает вероятность создания дрянных программ.

Простой пример. Допустим, что есть некоторая программа, которая должна скопировать файл с одного компьютера на другой. На платформе Windows прежний «прозрачный» способ это сделать заключался в вызове

---

[1] Подробнее об RPC см. *searchwebservices.techtarget.com/sDefinition/0„sid26_gci214272,00.html*.

[2] Подробнее о DCOM см. *www.microsoft.com/com/tech/dcom.asp*.

обычного метода CopyFile с использованием UNC-имен файлов вида \\SERVER\SHARE\Filename.

Если с сетью все в порядке, то метод работает прекрасно. Но если файл размером с мегабайт, а подключение к сети осуществляется через модем, то возникают всякие неприятности. Все приложение полностью замирает на время передачи мегабайтного файла. Индикатор выполнения показать нельзя, потому что при создании CopyFile предполагалось, что он всегда будет работать «быстро». После случайного разрыва соединения возобновить передачу нельзя.

На практике, если требуется передача файла по сети, лучше воспользоваться API типа FtpOpenFile и родственными функциями. Да, это не то же самое, что локальное копирование файла, и работать в этом случае сложнее, но тот, кто делал эту функцию, понимал, что сетевое программирование отличается от локального, и поэтому в функции реализованы ловушки для индикатора выполнения, корректного завершения при отсутствии или разрыве сетевого соединения и асинхронного выполнения.

Вывод: когда в очередной раз кто-нибудь попытается продать вам программный продукт, позволяющий обращаться к сетевым ресурсам так же, как к локальным, бегите от него со всех ног.

ГЛАВА ВОСЕМНАДЦАТАЯ

# Бикультурализм

14 ДЕКАБРЯ 2003 ГОДА, ВОСКРЕСЕНЬЕ

К данному моменту у Windows и UNIX функционально стало больше сходства, чем различий. Здесь и там поддерживаются одни и те же основные метафоры программирования, от командной строки до GUI и веб-серверов; они построены вокруг примерно одной и той же совокупности системных ресурсов, включая почти идентичные файловые системы, память, сокеты, процессы и потоки. Базовый набор сервисов каждой из этих операционных систем почти не ограничивает разнообразия создаваемых приложений.

Остались лишь культурные различия. Да, мы все принимаем пищу, но они там деревянными палочками едят сырую рыбу с рисом, а мы здесь едим руками куски молотой коровы на хлебе. Культурные различия не означают, что американские желудки не могут переваривать суши или что японские желудки не могут переварить биг-мак, и не означают также, что не может быть множества американцев, которые едят суши, или множества японцев, которые едят бургеры, но они означают, что у американца, впервые спустившегося по трапу самолета в Токио, возникает ошеломляющее чувство, что это место *необычно*, и никакое философствование о том, что *в глубине мы все одинаковы, мы любим, работаем, поем и умираем*, не может отменить того обстоятельства, что американцы и японцы никогда не смогут *по-настоящему* привыкнуть к тому, как устроены туалеты в другой стране.

Каковы культурные различия между программистами UNIX и Windows? Есть много деталей и тонкостей, но по большей части они сводятся к одно-

му: в культуре UNIX ценится код, который полезен другим программистам, а в культуре Windows ценится код, который полезен непрограммистам.

Конечно, это большое упрощение, но ведь это действительно большая разница, для кого мы программируем – для программистов или конечных пользователей! Все остальное – комментарии.

Нередко провоцирующий полемику Эрик С. Рэймонд (Eric Raymond) написал большую книгу под названием «The Art of UNIX Programming» (Addison-Wesley, 2003)[1], подробно исследовав в ней эту культуру. Можно купить бумажное издание и прочесть его либо, если взгляды и принципы Рэймонда слишком «антиидиотарны» для вас[2] и вы не хотите платить ему денег, почитайте ее в Интернете[3] бесплатно, сохранив уверенность, что автор не получит от вас ни копейки за свой тяжелый труд.[4]

Рассмотрим простой пример. В культуре программирования под UNIX высоко ценятся программы, которые могут вызываться из командной строки, принимать аргументы, контролирующие самые различные аспекты их поведения, и результат работы которых может быть получен в виде регулярно форматированного обычного текста, доступного для чтения машиной. Такие программы ценятся, потому что программистам легко включать их в другие программы или более крупные программные системы. Один маленький пример: в культуре UNIX есть важная ценность, которую Рэймонд называет «молчание – золото», означающая, что, если программа успешно выполнила в точности то, что вы ей сказали, она не должна вообще ничего выводить.[5] Неважно, что вы сделали – ввели команду из 300 символов для создания файловой системы, или скомпилировали и установили сложный программный пакет, или отправили пилотируемый космический корабль на Луну. Если выполнение прошло успешно, считается, что не надо выводить ничего. Если снова появляется подсказка командной строки, пользователь заключает из этого, что все OK.

---

1 Raymond, Eric (Эрик Рэймонд) «Искусство программирования для UNIX», Вильямс, 2005.

2 Raymond, Eric «Draft for an Anti-Idiotarian Manifesto (version 2)» (Проект антиидиотарианского манифеста (версия 2)), Armed and Dangerous, October 16, 2002. См. *armedndangerous.blogspot.com/2002_10_13_armedndangerous_archive.html#83079307*.

3 См. *www.faqs.org/docs/artu/*.

4 Raymond, Eric «Eric S. Raymond – Surprised By Wealth», Linux Today, December 10, 1999. См. *linuxtoday.com/news_story.php3?ltsn=1999-12-10-001-05-NW-LF*.

5 Raymond, Eric «The Art of UNIX Programming», Addison-Wesley, 2003, p. 20.

Это важная ценность в культуре UNIX, потому что вы программируете для других программистов. Как отмечает Рэймонд, «болтливые программы обычно плохо взаимодействуют с другими программами». Напротив, в культуре Windows вы программируете для тетушки Мардж, и тетушку Мардж можно оправдать, если она считает, что программу, которая ничего не вывела, потому что успешно отработала, нельзя отличить от программы, которая ничего не вывела, потому что произошел сбой, или программы, которая ничего не вывела, потому что не поняла ваш запрос.

Аналогично в культуре UNIX ценятся программы, не выходящие из текстового режима.[1] Там недолюбливают GUI, если не считать раскраски, явно нанесенной поверх текстового режима, и не любят двоичные форматы файлов. Это вызывается тем, что программировать взаимодействие с текстовым интерфейсом гораздо проще, чем, скажем, взаимодействие с GUI, что почти невозможно делать в отсутствие некоторых прочих условий, типа встроенного языка сценариев. Здесь мы снова сталкиваемся с тем, что культура UNIX ценит код, полезный другим программистам, что редко бывает целью при программировании в Windows .

Это не значит, что все UNIX-программы разрабатываются исключительно для программистов. Далеко не так. Но *культура* ценит то, что полезно программистам, и этим кое-что объясняется.[2]

Предположим, что у нас есть UNIX-программист и Windows-программист и оба они получили задание создать одинаковое приложение для конечного пользователя. UNIX-программист создаст базовое приложение командной строки или с текстовым интерфейсом и, может быть, с некоторым опозданием и нехотя построит GUI над этим базовым приложением. В результате основные операции этого приложения станут доступны другим программистам, которые смогут вызывать программу из командной строки и получать ее результаты в текстовом виде. Программист Windows скорее всего начнет с GUI и, может быть, с некоторым опозданием добавит язык сценариев, автоматизирующий действие GUI. Это естественно для

---

1 Raymond, Eric «The Art of UNIX Programming», Addison-Wesley, 2003, p. 105 («Искусство программирования для UNIX», Вильямс, 2005).

2 Кое-что, но далеко не все... Тот же интерфейс текстовой командной строки, который автор обсуждает двумя абзацами выше, предоставляет на порядок больше «степеней свободы», чем любой другой известный пользовательский интерфейс... В конце концов, нельзя всерьез пытаться сравнить на четырех страницах текста две «глобальные» культуры в мире IT. – *Примеч. науч. ред.*

культуры, в которой 99,999% пользователей никоим образом не являются программистами и не испытывают желания стать ими.

Есть большая группа Windows-программистов, которые пишут код преимущественно для других программистов: это собственно команда Windows внутри Microsoft. Они стремятся к тому, чтобы создать API для вызова из языка C, реализующий функциональность, а затем создавать GUI-приложения, обращающиеся к этому API. Любое действие, которое можно осуществить из интерфейса пользователя Windows, можно выполнить также через интерфейс программирования, вызываемый практически из любого языка программирования. Например, собственно Microsoft Internet Explorer представляет собой лишь крошечную программу размером 89 Кбайт, служащую оболочкой для десятков очень мощных компонентов, свободно доступных квалифицированным Windows-программистам и в большинстве своем обладающих гибкостью и мощью. К сожалению, поскольку программистам недоступен исходный код этих компонентов, последние можно использовать только такими способами, которые были предварительно рассмотрены и разрешены разработчиками компонент в Microsoft, что не всегда может удовлетворить другого программиста. Кроме того, иногда обнаруживаются ошибки, в которых обычно виноват тот, кто обращается к API, и которые трудно или невозможно отладить в отсутствие исходного кода. Тенденция UNIX-культуры предоставлять исходный код делает ее средой, для которой легче разрабатывать приложения.[1] Любой разработчик приложений для Windows может рассказать вам, как он потратил четыре дня на исправление ошибки, потому что полагал, что объем памяти, возвращаемый функцией LocalSize, будет таким же, как размер памяти, которую он изначально запросил в LocalAlloc, или про какую-нибудь другую ошибку, которую он исправил бы в десять минут, будь ему доступен исходный код библиотеки. В качестве иллюстрации Рэймонд сочиняет занятную историю, в которую вполне может поверить всякий, кто имел дело с библиотеками в двоичном формате.[2]

Теперь вы видите, в чем суть этих религиозных споров. UNIX лучше, потому что при отладке можно зайти в библиотеки. Windows лучше, потому что тетушка Мардж может увидеть подтверждение того, что ее почта действительно отправлена. На самом деле ни та, ни другая системы не *лучше*. В них просто разные системы ценностей. В UNIX главная ценность – облег-

[1] Raymond, Eric «The Art of UNIX Programming», Addison-Wesley, 2003, p. 379.

[2] там же, p. 376.

чить жизнь другим программистам, а в Windows главная ценность – облегчить жизнь тетушке Мардж.

Рассмотрим еще одно культурное различие. Рэймонд пишет: «Классическая документация UNIX пишется телеграфным стилем, но она полная... Этот стиль предполагает активного читателя, способного сделать очевидные и неупомянутые выводы из сказанного и обладающего достаточной уверенностью в себе, чтобы доверять этим выводам. Внимательно читайте каждое слово, потому что едва ли вам дважды станут повторять одно и то же». Боже мой, подумал я, ведь фактически он *рекомендует молодым программистам продолжать писать никому не понятные страницы руководства*.

С конечными пользователями такое не проходит. Рэймонд может называть это «сверхупрощающим высокомерием», но в культуре Windows признано, что конечные пользователи не любят читать,[1] и если они снизойдут до того, чтобы прочесть вашу документацию, то прочтут только самое необходимое, и вам придется давать объяснения еще и еще раз... Отличительной чертой хорошего файла подсказки Windows является то, что средний читатель может изучать любую тему в отдельности, для чего ему не требуется знание каких-либо других тем.

Как же получилось, что базовые ценности оказались разными? Это еще одно достоинство книги Рэймонда: он уходит глубоко в историю и эволюцию UNIX и вооружает молодых программистов знанием всего исторического опыта культуры, начиная с 1969 года. Когда создавалась UNIX и формировались ее культурные ценности, *конечного пользователя не существовало*. Компьютеры были дороги, процессорное время было дорого, и знания о компьютерах сводились к тому, как программировать. Неудивительно, что образовавшаяся культура ценила то, что было полезно другим программистам. Напротив, Windows создавалась с единственной целью: с выгодой продать как можно больше экземпляров. Мириады экземпляров. «Компьютер на каждом рабочем столе и в каждом доме» – вот что было прямой задачей разработчиков Windows, определяло их повестку дня и базовые ценности. Легкость использования непрограммистами была единственным способом получить доступ на каждый рабочий стол и в каждый

[1] Spolsky, Joel «User Interface Design for Programmers» (Проектирование интерфейса пользователя для программистов), Apress, 2001. См. *www.joelonsoftware.com/uibook/chapters/fog0000000062.html*.

дом, и потому «юзабилити ьber alles» стало культурной нормой. Программисты как целевая группа рассматривались на самом последнем месте.

Культурный раскол настолько глубок, что UNIX, по сути, никогда не претендовала на настольные системы. Тетушка Мардж фактически не сможет пользоваться UNIX, а неоднократные попытки создать для UNIX красивый интерфейс, с которым тетушка Мардж *сможет* работать, полностью провалились, потому что эти попытки делались программистами, насквозь пропитанными UNIX-культурой. Например, UNIX поддерживает принцип разделения политики и механизма,[1] исторически идущий от разработчиков X.[2] Он прямо привел к расколу в пользовательских интерфейсах; договориться во всех деталях о том, каким должен быть UI настольной машины, толком никогда не могли, *и считается, что это нормально*, потому что в этой культуре ценится разнообразие. А вот для тетушки Мардж совершенно *неприемлемо* использовать в разных программах разные интерфейсы для копирования и вставки. Вот таково положение через 20 лет после того, как разработчики UNIX начали попытки приладить к своим системам приличный интерфейс пользователя, когда глава крупнейшего поставщика Linux высказывает мнение, что домашним пользователям следует работать в Windows.[3] Я слышал утверждения экономистов о том, что Кремниевую долину невозможно было бы воспроизвести, скажем, во Франции, потому что французская культура так жестоко наказывает неудачников, что предприниматели стараются не рисковать. Возможно, то же справедливо в отношении Linux: она может никогда не стать настольной операционной системой, потому что в ее культуре ценятся вещи, препятствующие этому. Доказательством служит OS X: компания Apple все-таки создала UNIX для тетушки Мардж, но только потому, что инженеры и менеджеры Apple были прочно связаны с культурой конечного пользователя (которой я захватнически присвоил название «культуры Windows», хотя исторически она возникла в Apple). Они отказались от фундаментальной нормы культуры UNIX – построения вокруг интересов программиста. Они даже переименовали главные каталоги – полная ересь! – и стали употреблять обычные английские слова, например «applications» и «library» вместо «bin» и «lib».

---

1 Raymond, Eric «The Art of UNIX Programming», Addison-Wesley, 2003, p. 16–17.

2 *См. wwwx.org/.*

3 Kotadia, Munir (Мунир Котадиа) «Red Hat recommends Windows for consumers» (Red Hat рекомендует Windows потребителям), ZDNet UK, November 4, 2003. См. *news.zdnet.co.uk/software/linuxunix/0,39020390,39117575,00.htm.*

Рэймонд делает попытку сравнить и противопоставить UNIX и другие операционные системы, и это самая слабая часть прекрасной в остальных отношениях книги, потому что он не знает предмета, о котором ведет речь. Как только он заговаривает о Windows, обнаруживается, что источником его знаний программирования для Windows оказываются по большей части газеты, а не опыт программирования в Windows. Ну и пусть, он – не Windows-программист, и простим ему это. Как это характерно для человека, глубоко знающего одну культуру, он знает, что ценится в его культуре, но плохо различает те части его культуры, которые универсальны (убиение старушек, аварийное завершение программ: *всегда плохо*), и те, которые относятся только к программированию для программистов (сырая рыба, аргументы командной строки: *смотря для кого*).

Есть очень много монокультурных программистов, которые, как типичный американский парень, никогда не выезжавший за пределы Сент-Пол, штат Миннесота, не ощущают разницы между ценностями, принадлежащими некой культуре, и базовыми человеческими ценностями. Я очень часто встречал UNIX-программистов, издевавшихся над программированием в Windows и считавших Windows варварством и глупостью. Рэймонд слишком часто попадает в ловушку пренебрежительного отношения к ценностям других культур без учета их истоков. Такой фанатизм довольно редко встречается среди программистов Windows, которые в целом ориентированы на решение задачи и свободны от идеологических предрассудков. Во всяком случае Windows-программисты признают недостатки своей культуры и прагматически говорят: «Видите ли, чтобы продать текстовый процессор как можно большему числу людей, его надо научить выполняться на их компьютерах, а если из-за этого приходится хранить настройки в Гнусном Реестре, а не в элегантных файлах ~/.rc, то тут уж ничего не поделаешь». Мир UNIX пышет самодовольством культурного превосходства, пропаганды и поклонения карме Slashdot, тогда как мир Windows более практичен («да, что делать, надо же на жизнь зарабатывать»), и поэтому культура UNIX чувствует себя осажденной, не способной вырваться из серверной стойки и области хобби на широкие просторы настольных машин. Это высокомерие слабости – самый большой недостаток книги «The Art of UNIX Programming», хотя сам по себе он не столь велик. В целом в книге настолько много чрезвычайно интересного и глубокого проникновения во многие аспекты программирования, что можно вытерпеть отдельные дурно попахивающие идеологические эскапады ради возможности многое почерпнуть из нее относительно всеобщих идеалов. Я действи-

тельно рекомендую эту книгу разработчикам, к какой бы культуре они ни принадлежали, на любой платформе, с любыми задачами, потому что очень многие из провозглашаемых в ней ценностей являются универсальными. Когда Рэймонд указывает, что формат CSV хуже, чем формат */etc/passwd*, он думает тем самым добавить очков UNIX против Windows,[1] но знаете что? Он прав. */etc/passwd* действительно *проще* разбирать, чем CSV, и, прочтя его книгу, вы узнаете почему и станете лучше как программист.

[1] Raymond, Eric «The Art of UNIX Programming», Addison-Wesley, 2003, p. 109.

ГЛАВА ДЕВЯТНАДЦАТАЯ

# Отчеты об авариях от пользователей – автоматически![1]

18 апреля 2003 года

Одно время я работал разработчиком клиентского программного обеспечения для очень крупного интернет-провайдера в США. Наше программное обеспечение использовали буквально *миллионы* людей. Даже очень редко проявляющаяся ошибка могла повлиять на работу сотен и тысяч пользователей. В бета-версии самого последнего релиза мы применили технологию автоматического сбора информации об отказах, когда реальные отчеты об отказах обрабатывались и заносились в базу данных, позволяя разработчикам обнаруживать и исправлять ошибки, проявлявшиеся только в реальной работе. Обычно мы не могли отловить такие ошибки на испытательном стенде, поскольку невозможно воспроизвести каждую необычную конфигурацию машины, которая может оказаться у пользователя. Поэтому я чувствовал себя очень уверенно, когда мы решили дать отмашку и выпустить свой новейший код. Помню, я сказал своему отцу: «Бета выглядит отлично; вчера у нас было всего двенадцать отказов по всей Северной Америке».

Двенадцать? А почему не тринадцать?

Нет. Двенадцать. Уверенность, обретаемая благодаря знанию о *любом отказе*, в какой бы точке света он ни произошел, имеет очень большое значение для выпуска высококачественного продукта, предназначенного для работы в неконтролируемой среде. Когда вы создаете программное обеспечение, доступное всем желающим, нельзя рассчитывать на то, что

[1] Впервые опубликовано в журнале «Software Testing & Quality Engineering».

клиенты станут сообщать вам о фатальных сбоях вашей программы – одни недостаточно подготовлены технически, а остальные просто не станут тратить свое драгоценное время, чтобы сообщить вам полезные сведения о возникшем отказе, если только вы не сделаете процедуру такого сообщения полностью автоматизированной.

Сейчас у меня собственная компания, и я принял меры, чтобы практически любой наш код автоматически сообщал об ошибках команде разработчиков. Даже те программы, которые мы пишем исключительно для внутреннего применения, извещают разработчиков о случаях возникновения аварийной ситуации. Я хочу рассказать всем о выработанных на протяжении ряда лет методах и полученных уроках, связанных со сбором сообщений о сбоях при реальной эксплуатации программ.

## *Сбор данных*

Итак, ваш код аварийно завершился. Практически в каждой программной среде есть централизованные способы восстановления после фатального сбоя (см. примеры на врезках). В этом случае мы не позволяем программе просто так скончаться, а показываем диалоговое окно автоматизированного доклада, краткого и по существу, о возникновении фатального сбоя:


Если задать лишь один вопрос, есть вероятность, что пользователь не поленится ответить на него.

Ввод почтового адреса должен быть необязательным, чтобы уменьшить тревогу по поводу конфиденциальности.

Пользователю, озабоченному секретностью и анонимностью, надо дать возможность узнать, какие конкретные данные будут переданы.

С годами я уяснил, что чем больше вопросов вы задаете, тем меньше вероятность получить на них ответ. Поэтому мы задаем минимум вопросов, призванных помочь нам диагностировать проблему. Почти все остальные важные сведения, например версию операционной системы, объем памяти и т. д., обычно можно получить автоматически.

Необходимо уверить пользователя в анонимности и секретности этого сообщения о сбое. Те, кто работает с конфиденциальными данными, могут не пожелать передавать сообщение о сбое, если заподозрят, что мы хотим заполучить себе все их секретные данные, поэтому мы указываем ссылку, по которой пользователи могут щелкнуть и узнать, что конкретно мы хотим передать. Чтобы избежать даже подозрения в нечестности, постарайтесь указать также, какую автоматически полученную информацию вы хотите передать.

Следующий вопрос: какие данные, полезные разработчикам для выяснения причин аварии, надо собрать автоматически? Существует соблазн схватить *все* – любую доступную системную информацию: версии всех DLL и элементов COM, а то и полный дамп ядра.

Проработав несколько лет разработчиком, я так и не получил четкого представления о том, что же мне делать с дампом ядра, и пришел к выводу, что в получении этих данных, в сущности, нет необходимости. Мы пришли к выводу, что знание строки кода, в которой произошел фатальный сбой, в большинстве случаев дает достаточно информации. В тех редких случаях, когда этой информации недостаточно, можно связаться с пользователями, у которых возник этот сбой, по электронной почте и запросить дополнительные сведения, которые могут быть полезны.

Благодаря тому, что вы собираете так мало информации, процедура отправки сообщения о сбое проходит очень быстро и меньше раздражает пользователей. Одно лишь получение номеров версий всех DLL и элементов COM может занять значительное время, особенно если учесть время передачи по модемной линии, а полезная информация при этом обнаруживается очень редко. Даже если выяснится, что некий фатальный сбой происходит только в присутствии определенной версии системной DLL Microsoft, что вы сможете сделать? Вам все равно придется исправлять код, чтобы избежать этого сбоя.

*Данные, которые мы собираем автоматически*

- Точная версия нашего продукта
- Версия ОС и версия Internet Explorer (очень многие части Windows обеспечиваются Internet Explorer и его компонентами, поэтому важно знать это даже для приложений GUI)
- Файл и номер строки кода, где произошел сбой
- Сообщение об ошибке в виде строки
- Уникальный числовой код для такой ошибки
- Описание пользователем выполнявшихся им действий
- Адрес электронной почты пользователя

Если пользователь сообщает свой почтовый адрес, разработчик может нажать в базе данных кнопку «Ответить» и тут же послать сообщение с запросом дополнительной информации. База данных автоматически хранит в сообщении об ошибке копии всех входящих и исходящих сообщений, касающихся этой ошибки.

## *Звонок домой*

Интернет вездесущ, поэтому практически всегда лучше всего направлять информацию через Веб. Если отправить ваше сообщение об ошибке в виде стандартного запроса http, оно пройдет практически любой сетевой экран, который может быть установлен у пользователя. Сейчас фактически в любой программной среде есть встроенные библиотеки для отправки запроса HTTP и получения ответа. Например, в Windows есть встроенные функции библиотеки WININET, которые используют код Explorer для передачи по сети при отправке запроса HTTP и получении ответа. Самое приятное в этих функциях то, что, даже если пользователь сконфигурировал свой броузер так, чтобы запросы направлялись на прокси-сервер, что обычно случается за брандмауэрами, обращения к WININET автоматически пройдут через прокси-сервер без всяких дополнительных усилий с вашей стороны.

Наш сервер возвращает очень короткий файл XML, указывающий, что сообщение об ошибке получено, и содержащий некоторое сообщение для пользователя. Если ваше приложение размещено в веб, можно поступить даже проще: вывести веб-страницу с формой, которая передаст данные на ваш сервер.

Иногда вместо немедленной отправки сообщения о сбое лучше записать сообщение в файл или реестр и послать его, когда пользователь в следующий раз запустит программу. Я называю такую технологию *отложенной передачей*. Хотя в результате отчет будет получен позднее, но есть и преимущество: если сбой был настолько серьезен, что приложение оказалось не в состоянии передать сообщение, вы все равно его получите.

Все сообщения об ошибках приходят в Fog Creek через единственный URL на нашем открытом для внешнего доступа сервере. Наша база данных учета ошибок получает сообщения об ошибках через этот единственный URL. Фактически этот URL предоставляет единственный способ доступа к нашей базе данных извне: все остальное закрыто, поэтому посторонние могут передавать сообщения об ошибках, но не могут получить доступ к базе данных. Вот как выглядит сообщение об ошибке в FogBUGZ:


Можно было бы с помощью соответствующих настроек направлять сообщение об ошибке кому-нибудь из бригады разработчиков, но мы предпочли создать виртуального пользователя «CityDesk New Bug» (Новый баг в CityDesk). Мы регулярно просматриваем все ошибки, адресованные этой виртуальной личности, и решаем, надо ли их исправлять. Те ошибки, которые решено исправить, назначаются конкретному лицу.

## *Выявление дубликатов сбоев*

Важная особенность автоматизированного сбора сообщений об ошибках состоит в том, что они могут многократно появляться у одних и тех же или разных людей, поэтому нежелательно вводить в базу данных новые записи для каждого *дубликата* сбоя. Мы решаем эту проблему путем создания уникальной строки, содержащей главные элементы данных об ошибке. Когда обнаруживаются сбои с одинаковой уникальной строкой, они добавляются к уже зарегистрированному случаю, не создавая нового. Благодаря этому программист может в одном месте наблюдать все случаи, дублирующие некоторый сбой.

Мы стараемся составлять эту строку так, чтобы для двух людей, у которых произошел один и тот же сбой, она была одинакова. Поэкспериментировав, мы пришли к выводу, что лучше всего включать в нее номер ошибки, имя файла, название функции, номер строки и версию нашей программы. В приведенном скриншоте FogBUGZ уникальная строка такова: **Error 91 (global:IsRoot:0) V1.0.32**. Она означает, что ошибка с номером 91 произошла в файле с именем *global.bas* в функции `IsRoot` в строке 0, выполнявшейся в версии 1.0, сборка 32. Кстати, мы всегда задаем четные номера для внутренних сборок, не рассылаемых клиентам, и нечетные номера для всех сборок, которые попадают клиентам, поэтому я могу мгновенно определить, что данный сбой произошел у разработчика, а не у клиента.

Разработать формат уникальной строки не так просто. В прошлом мы включали в эту строку весь текст сообщения об ошибке. Однако вскоре выяснилось, что сообщение об ошибке переведено на разные языки. Поэтому каждое сообщение об ошибке мы получали отдельно на английском, немецком, испанском, французском и Бог весть на каких еще неизвестных мне языках! Мы вышли из положения, поместив *текст* ошибки в *тело* сообщения, а *номер* ошибки, который не зависит от языка, – в *заголовок*, избавившись в результате от массы дубликатов.

Заголовки мы тоже форматируем так, чтобы по ним можно было легко искать конкретные проблемы. Благодаря формату заголовка `имя_файла:функция:номер_строки` (обратите внимание на двоеточия) легко найти ошибки в конкретной функции путем поиска «:функция:» По той же причине мы ставим V перед номером версии: можно искать V1, или V1.0, или V1.0.32. Если бы не было V, то при поиске версии 1 мы получали бы все отчеты, где в любом месте заголовка есть число 1.

После того как ошибка идентифицирована, можно поменять флажок «Продолжать получение сообщений» на «Прекратить получение сообщений», в результате чего сообщения о сбоях с такой же уникальной строкой будут просто игнорироваться. Можно даже задать текст сообщения, которое начнет рассылаться всем пользователям, у которых возникнет такой же сбой. Мы предлагаем в них найденные выходы из ситуации, например: «Когда в следующий раз соберетесь сохранять данные, сначала похлопайте себя по голове и почешите живот!»

Часто причина появления дубликатов сообщений о сбоях кроется в самом коде обработки сбоев. Это *не обязательно* связано с ошибками в коде обработки сбоев – просто первоначальный сбой мог так сильно все покалечить, что *никакой* код не мог после этого нормально работать.

## *Отладка*

Во время бета-тестирования мы стараемся сразу изучать каждое сообщение о сбое, однако когда продукт уже выпущен и разработчики трудятся над следующей версией, у них обычно нет времени, чтобы рассматривать все поступившие сообщения. Поэтому мы обычно наблюдаем в течение нескольких месяцев, какие сбои появляются чаще всего, и работаем только над ними. Недостаток в том, что едва ли удастся связаться с пользователем, чтобы задать ему вопросы о сбое, который случился у него несколько месяцев назад: он просто не будет помнить подробности. Но я заметил, что если один и тот же сбой происходил несколько раз, то один из пользователей всегда даст достаточно информации о том, какие действия он выполнял, чтобы я смог воспроизвести его сбой у себя. Фактически, зная, в какой строке кода произошел сбой, почти всегда удается выяснить, в чем примерно может быть проблема, даже если трудно придумать, как воспроизвести ситуацию. Однажды я буквально выполнил код в обратном направлении с помощью арифметики и логических рассуждений, пытаясь воспроизвести ситуацию. Если здесь произошел сбой, рассуждал я, тогда это значение отрицательно. Если оно отрицательно, тогда этот оператор `IF` должен быть истинным. И так далее, пока я не установил, какое сочетание значений привело к сбою, и не понял, откуда оно возникло.

## *Назначение приоритетов*

Если вы создали систему автоматизированного информирования об ошибках, то какой бы надежной ни казалась ваша программа, вы нач-

нете получать довольно значительный поток сообщений о сбоях. Поэтому большую важность приобретает умение сортировать этот поток: выбрать те ошибки, которые необходимо исправлять, и игнорировать остальные.

Изучая получаемые сообщения, я обычно обращаю внимание на признаки тех ошибок, которые мы вряд ли когда-нибудь исправим. Например:

- У пользователя сбои в компьютере или некачественная память.
- Пользователь экспериментирует, изменяя наши файлы.
- У пользователя стоит старая операционная система типа Windows 95, способствующая возникновению сбоев.
- Пользователь выполняет программу в условиях чувствительной нехватки памяти – вероятно, с полным диском и полным ОЗУ.

Иногда просто невозможно понять, что вызвало сбой, особенно если он произошел *однократно*. На самом деле я придерживаюсь правила даже *не рассматривать* сбой, если он случился один раз. У нас есть рыба покрупнее. Если ошибка не воспроизводится в практических условиях, вряд ли ее удастся воспроизвести на стенде.

## *А зачем вам это?*

Итак, вы представляете себе суть работы с автоматизированной системой сообщений об ошибках и можете решить, нужна ли она вам. Ответ на этот вопрос в некоторой степени зависит от того, кто ваши пользователи.

Если вы разрабатываете программное обеспечение для применения в своей же фирме, то автоматические сообщения о сбоях едва ли вам понадобятся. Внутрифирменные программы часто пишут для решения какой-то конкретной проблемы, и они обходятся гораздо дороже, чем готовые продукты. Если проблема решена, не стоит больше тратить деньги на этот проект. Если в коде происходит сбой раз в неделю, это досадно, но с точки зрения бизнеса тратить несколько тысяч долларов на то, чтобы разработчик исправил ошибку, может быть неоправданным. Это, может быть, *красиво*, но *невыгодно*. Программисты-идеалисты часто бывают недовольны тем, что их руководители предлагают им прекратить работу над кодом, как только он становится «достаточно хорош». Код решает проблему бизнеса, хотя его качество могло быть выше, а любая дополнительная работа над ним экономически невыгодна. Однако многие корпоративные разработчики вынуждены работать фактически без всякого контроля качества и специальных отладчиков, поэтому система автоматизированного сбора информации о сбоях может быть единственным способом узнавать об ошибках.

С другой стороны, если вы разрабатываете программное обеспечение на продажу или для заказчиков, то качество оказывается важным фактором конкурентной борьбы. И вашим программам придется работать во враждебной среде. С машинами, стоящими у клиентов, всегда *неразбериха*. Среди них не найдется двух одинаковых. У всех есть какие-нибудь различия в аппаратной или программной конфигурации. Компании-поставщики устанавливают на PC самые невообразимые программы. Многие покупатели радостно загружают и устанавливают у себя на машине все, что им приглянется, в том числе разные очень «умные» утилиты, которые внедряются в пространства процессов других работающих приложений. Большинство домашних пользователей недостаточно разбираются в компьютерах, чтобы правильно наладить их работу. В такой враждебной среде автоматизированный сбор сообщений об ошибках представляет собою единственный способ достичь требуемого рынком уровня качества. Создание надежной системы, которая будет обрабатывать сбои, возникающие в реальных условиях, сообщать о них, классифицировать и вести учет, удовлетворит ваших клиентов и многократно окупится высоким качеством поставляемого вами кода.

## *Обработка сбоев*

### *Обработка сбоев в коде Visual Basic*

Поскольку в обычной управляемой событиями программе Visual Basic 6.0 очень много точек входа (по одной для каждого обрабатываемого события), единственный способ перехватить сбой, возникший в *любом* месте кода, – это организовать обработку ошибок в *каждой* функции. Вот как выглядят *все* наши функции:

```
Private Sub cmd_Click()
On Error GoTo ERROR_cmd_Click

   ... фактический код функции ...
   Exit Sub

ERROR_cmd_Click:
   HandleError "moduleName", "cmd_Click"
End Sub
```

Писать весь этот дополнительный код обременительно. К счастью, есть утилита ErrorAssist (*http://www.errorassist.com*), которая автоматически добавляет код для перехвата ошибок во все ваши функции. В каждом случае мы вызываем глобальную функцию HandleError, которая выводит диалоговое окно сообщения об ошибке.

### *Обработка сбоев в коде Windows API*

В Win32 API есть понятие *структурной обработки исключительных ситуаций*. Когда происходит сбой, Windows ищет текущую функцию-обработчик необработанной исключительной ситуации и вызывает ее. Если функции, которая разберется с такой исключительной ситуацией, нет, выводится обычное недружелюбное окно «Программа выполнила недопустимую операцию».

Чтобы установить свою функцию необработанной исключительной ситуации, нужно сделать две вещи. Сначала реализовать собственную функцию вида

```
LONG UnhandledExceptionFilter(
 STRUCT _EXCEPTION_POINTERS *ExceptionInfo
);
```

Затем нужно вызвать функцию SetUnhandledExceptionFilter, передав ей указатель на вашу функцию UnhandledExceptionFilter.

В коде C++ то же самое можно сделать, если заключить главную точку входа в конструкцию __try/__catch. Обратите внимание на удвоенные символы подчеркивания, которые заставляют компилятор обрабатывать структурированные исключительные ситуации, генерируемые отказами на низком уровне, типа разыменования нулевых указателей, а не самодельные исключительные ситуации C++, которые вы генерируете сами и перехватываете с помощью try и catch.

Подробнее читайте в разделах «Using Structured Exception Handling» Windows Platform SDK или MSDN.[1]

[1] Эти вопросы рассмотрены в книге Дж. Рихтера «Windows для профессионалов. Создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows», СПб.: Питер, 2001. – *Примеч. науч. ред.*

### *Обработка сбоев в приложениях ASP*

Microsoft Internet Services Manager позволяет задать собственную страницу обработки ошибок в виде HTML или ASP, которая будет обрабатываться для любых ошибок сценария, не перехватываемых операторами `On Error`. В частности, при аварийном завершении приложения ASP или возникновении в нем какой-либо необрабатываемой ошибки страница переадресуется обработчику ошибок «500;100». Во всех приложениях у нас есть страница ASP для перехвата ошибок 500;100.

На этой странице находится такой главный код VBScript:

```
Set objASPError = Server.GetLastError
```

Теперь у вас будет объект с именем `objASPError`, содержащий массу полезных данных о последнем возникшем сбое, включая имя файла и номер строки.

Образец кода для обработки ошибок ASP устанавливается на любом компьютере вместе с IIS: поищите в каталоге *\windows\help\iishelp\common* файл с именем *500–100.asp*. Он просто показывает детали ошибки ASP конечному пользователю. Взяв файл *500–100.asp* за основу, можно создать свою специализированную страницу сообщения об ошибке, поместив на нее форму со скрытыми элементами, содержащими данные о сбое. В этой форме надо задать атрибут `action`, который отправит всю информацию об ошибке на другую веб-страницу.

ЧАСТЬ ВТОРАЯ

# Руководство разработчиками

ГЛАВА ДВАДЦАТАЯ

# Справочник бойца по проведению собеседования

4 июня 2004 года, пятница

Краткая викторина.

Пестрая шайка анархистов, защитников свободной любви и агитаторов за права бананов, захватила судно «The Love Boat» в Пуэрто Вальярта и угрожает потопить его через семь дней со всеми 616 пассажирами и 327 членами команды, если не будут выполнены их требования. Какие? Миллион долларов мелкими купюрами и версия WATFIV, известного компилятора Waterloo Fortran IV, с лицензией GPL. (Удивительно, как мало общего нашлось у защитников свободной любви и агитаторов за права бананов.)

Как главный программист отдела программирования в «Праздничных круизах» вы должны решить, сможете ли вы написать компилятор Fortran с нуля за семь дней. В вашем распоряжении два программиста.

– Сможете сделать это?

– Ну, это зависит... – говорите вы. Все-таки хорошо писать книги – я могу вложить в ваши уста любые слова, и вы будете бессильны хоть что-нибудь с этим поделать!

– От чего?

– Ну, от того, будут ли у нас средства генерации UML.

– Это действительно так важно? Три программиста, семь дней, Waterloo Fortran IV. Все решает наличие средств UML?

– Думаю, что нет.

– Хорошо, тогда от чего все зависит?

– А будут у нас 19-дюймовые мониторы? И неограниченное количество джолт-колы?

– Опять-таки, это важно? Без кофеина вам не справиться?

– Думаю, что неважно. Погодите, вы сказали, что у меня в штате два программиста? (Обратите внимание, становится интересно.)

– Верно.

– Кто?

– А это важно?

– Еще бы! Если мы не поладим друг с другом, то не сможем работать. И я знаю несколько суперпрограммистов, которые могут склепать компилятор Fortran за неделю *в одиночку*, а также *массу* программистов, которые не напишут код для печати начального заголовка и за полгода.

Вот теперь мы кое-что выяснили!

Все соглашаются с тем, что люди – самая главная часть программного проекта, но никто не знает толком, как *решать* эту проблему. Первое, что надо правильно сделать, если вы хотите, чтобы у вас были хорошие программисты, – это *нанять* правильных программистов. А это значит, что вы должны уметь определить, *какие* программисты правильные, что обычно делается во время собеседования. Итак, поговорим о собеседованиях.

Вот как обычно проходит подбор кадров. Сначала вы берете толстую пачку резюме. Проглядев их все, вы отбрасываете те, которые не кажутся вам блестящими. Возможно, я поступаю неправильно, но обычно я выкидываю резюме, в которых много ошибок, предполагая, что неряшливо написанное резюме означает неряшливость и в манере работы. (Однажды агент по найму подкинул мне резюме предполагаемого Windows-программиста, в котором операционная система была названа «MicroSoft Window».) Но важнее, чтобы резюме свидетельствовало о способности соискателя пройти процедуру жесткого отбора, из резюме должно быть видно, что он уже проходил такой отбор. Например, учеба в университете со строгим отбором. О некоторых компаниях, например, известно, что они очень придирчивы в найме работников; если в резюме есть названия таких компаний, может быть, стоит тщательнее присмотреться к кандидату. Хорошим признаком служат элитные воинские части со строгим отбором. То есть надо искать парашютистов, учившихся в Йейле и работавших в Microsoft – как я, например.

Конечно, нанимая людей, исходишь не только из этого.

Обычно следующим шагом является проверка по телефону: звонишь человеку и обсуждаешь с ним какую-нибудь проблему программирования в течение получаса. Например: «Как бы вы стали писать анализатор XML?»

Три четверти тех, с кем разговариваешь по телефону, оказываются не слишком сообразительными. Приходится повторять им одно и то же много раз. Собственных идей у них нет. Каждая клетка в вашем мозгу начинает кричать: идиот! Вы сэкономите время и деньги на личных беседах, если сразу отсечете таких слабоумных.

Наконец, личная беседа в рабочей обстановке. Всегда старайтесь, чтобы каждого кандидата на работу интервьюировало не меньше шести человек, причем не менее пяти из них должны иметь одинаковый с ним ранг (т. е. программисты, а не менеджеры). Вы ведь знаете компании, в которых каждого кандидата интервьюирует какой-нибудь старый опытный менеджер? В таких компаниях работают не самые лучшие люди. Слишком легко одурачить проверяющего во время одного интервью, особенно если непрограммист интервьюирует программиста.

Если хотя бы двое из беседовавших с кандидатом считают, что не стоит его нанимать, не берите его. Это означает, что формально вы можете завершить «день» интервью после первых же двух, если известно, что кандидат не будет нанят, что разумно. Но чтобы не быть жестоким, лучше не предупреждать кандидата о том, сколько интервью его ждет. Я слышал, что в некоторых компаниях любой интервьюер может отклонить кандидата. Мне кажется, что это уж слишком: я бы разрешил любому руководителю отклонять кандидата, но не стал бы отказывать кому-то лишь потому, что он не понравился одному младшему сотруднику.

Не пытайтесь одновременно интервьюировать целую группу людей. Это плохо. На каждом интервью должны быть один интервьюер и один интервьюируемый, комната с дверью, которую можно закрыть, и доска для рисования. По своему богатому опыту я могу сказать, что если интервью продолжается менее часа, то не чувствуешь себя в состоянии принять правильное решение.

Среди приглашенных на интервью попадаются три типа людей. На одном полюсе находится голытьба, у которой отсутствуют даже самые элементарные навыки, необходимые для этой работы. Их легко обнаружить и отчислить, как правило, задав два-три несложных вопроса. На другом полюсе находятся суперзвезды, которые напишут компилятор lisp на ассемблере для Palm Pilot за выходные просто для собственного удовольствия. А в середине находятся многочисленные «может быть», производящие впечатление, что они могут что-то сделать. Фокус в том, чтобы отличить суперзвезд от «может быть», потому что секрет в том, чтобы не принять «может быть». Ни в коем случае.

По окончании интервью вы должны быть готовы принять четкое решение относительно кандидата. Это решение может иметь только два вида: *принять* или *не принять*. Других ответов быть не должно. *Никогда* не говорите: принять, но не в мою бригаду. Это грубо и подразумевает, что кандидат недостаточно способен, чтобы работать у вас, но для неудачников в другой бригаде он может сгодиться. Если у вас возникает соблазн сказать «принять, но не в мою бригаду», просто замените автоматически текст на «не брать», и все будет в порядке. Даже если вам попался кандидат, который великолепно справится с вашей конкретной задачей, но не очень подойдет для других бригад, ответом должно быть *не брать*. В разработке ПО все меняется так быстро и часто, что нужны люди, способные успешно справиться практически с любой задачей, которую вы им поставите. Если каким-то образом вам попадется идиот, невероятно хорошо знающий SQL, но совершенно не способный освоить никакую другую тему, его *не брать*. Иначе вы решите краткосрочную задачу, но получите долгую головную боль.

*Никогда* не говорите «может быть, но точно не могу сказать». Если вы не можете сказать точно, значит, *не брать*. Это проще, чем может показаться на первый взгляд. Не уверены? Просто скажите «нет»! Если вы в затруднении, это означает *не брать*. Никогда не говорите: «Ну, пожалуй, берем, но меня немного тревожит, что...». Это тоже значит *не брать*. Просто механически переводите все колебания в «нет», и вы не ошибетесь.

Почему я здесь так непреклонен? Потому что гораздо, *гораздо* лучше отклонить хорошего кандидата, чем принять плохого. Плохой отнимет много денег и сил, а также время у других для исправления его ошибок. Увольнение нанятого по ошибке может отнять месяцы и оказаться очень сложным, если он будет склонен к сутяжничеству. Иногда просто невозможно кого-нибудь уволить. Плохие работники деморализуют хороших. При этом они могут быть плохими программистами, но *прекрасными людьми*, или им *очень нужна эта работа*, поэтому вам тяжело уволить их, или их увольнение вызовет всеобщее неудовольствие, а могут быть и еще причины. Очень тяжелая ситуация.

С другой стороны, отказывая хорошему кандидату, вы *чувствуете*, что в каком-то экзистенциальном смысле совершили несправедливость. Но если они действительно такие толковые, то не надо за них беспокоиться: они найдут *массу* хороших вариантов трудоустройства. Не бойтесь отвергнуть слишком много людей и никого не найти. Во время интервью это не ваша проблема. Конечно, важно найти хороших кандидатов. Но когда у вас реально находится кто-то на собеседовании, представьте себе, что за дверью

ждут еще 900 человек. Не снижайте свои требования, каким бы трудным вам ни казался поиск удовлетворяющих им кандидатов.

Все-таки самого главного я вам не сказал. Как же определить, что этого человека надо принять?

В принципе просто. Искать надо тех, кто:

1. Умен.
2. Доводит дело до конца.

Вот и все. Искать надо именно это. Запомните наизусть. Повторяйте это себе каждый день перед сном. У вас не хватит времени, чтобы выяснить большее в коротком интервью, поэтому не тратьте зря время, пытаясь выяснить, приятно ли было бы вам застрять с кандидатом в аэропорту или действительно ли он разбирается в программировании ATL и COM (или только делает вид).

Те, кто *умны*, но не *доводят дело до конца*, часто имеют докторские степени и работают в больших компаниях, где к ним никто не прислушивается, потому что они совершенно непрактичны. Они предпочитают рассуждать о теоретических аспектах, нежели вовремя выдавать результат. Такого рода люди выявляются по склонности находить теоретическое сходство между весьма различными понятиями. Например, они скажут, что «электронные таблицы, в сущности, представляют собой частный случай языка программирования», и после этого вы не увидите их неделю, пока они будут писать захватывающую и увлекательную статью о теоретических атрибутах вычислительной лингвистики электронных таблиц как языка программирования. Умно, но бесполезно. Другой признак этих людей – их склонность появляться в офисе с чашкой кофе в руке и начинать долгую беседу о сравнительных достоинствах интроспекции Java и библиотек типов COM *в тот самый день, когда вам надо выпускать бета-версию*.

Те, которые *доводят дело до конца*, но *не умны*, делают глупые вещи, очевидно, не подумав, а потом кому-то другому приходится подчищать то, что они наделали. В результате они становятся *убытком* для компании, потому что не только не вносят вклада в ее продукцию, но и отнимают время у добрых людей. Люди такого типа копируют большие куски кода, вместо того чтобы написать подпрограмму, и этим решают задачу, но не самым толковым образом.

Как можно определить во время интервью, что человек *умен*? Прежде всего на это указывает то, что не приходится объяснять одно и то же многократно. Беседа течет непринужденно. Часто кандидат говорит нечто, демонстрируя подлинное понимание, ум, сообразительность. Поэтому важ-

ная часть интервью – создание ситуации, в которой кандидат может показать, насколько он умен. Худший тип интервьюера – хвастун. Он беспрерывно болтает и почти не дает возможности кандидату сказать «да, это так, верно, я совершенно с вами согласен». Хвастуны принимают на работу всех: они считают, что кандидат умен, поскольку «думает совершенно, как я!»

Второй худший тип интервьюера – это шоумен, проводящий конкурс. Для него «умный» равносильно «много знает». Такой задает ряд тривиальных вопросов, касающихся программирования, и ставит очки за правильные ответы. Вот, для смеха, самый худший вопрос, который только можно задать на интервью: «Какая разница между varchar и varchar2 в Oracle 8i?» Ужасный вопрос. Нет никакой мыслимой корреляции между теми, кто знает ответ на этот конкретный мелкий вопрос, и теми, кого надо принять на работу. Не все ли равно, какая разница? Ответ можно найти в Интернете за 15 секунд! Помните, что «умный» и «знающий ответы на дурацкие вопросы» – это не одно и то же. Что бы вы ни говорили, компаниям-разработчикам ПО нужны люди со *способностями*, а не с конкретными навыками. Любые конкретные навыки, с которыми люди придут на работу, через пару лет устареют, поэтому лучше брать людей, которые смогут освоить любую новую технологию, а не тех, кто, *не задумываясь*, ответит, как организовать связь JDBC с базой данных MySQL.

Но в целом лучше всего можно узнать человека, дав ему возможность говорить. Задавайте открытые вопросы или ставьте задачи.

Так о чем же спрашивать?

## *Какие вопросы задавать на собеседовании*

Вопросы, которые задаю лично я, ведут свое происхождение от моей первой работы в Microsoft. Известны сотни знаменитых вопросов, которые задают на интервью в Microsoft. Каждый знает несколько вопросов, которые там очень любят. Со временем у вас тоже выработается конкретный список вопросов и стиль интервьюирования, помогающие принять решение «брать/не брать». Вот некоторые используемые мной приемы, показавшие свою успешность.

Перед интервью я читаю резюме кандидата и набрасываю план интервью на клочке бумаги. Это просто список вопросов, которые я собираюсь задать. Вот типичный план собеседования с программистом:

1. Знакомство.
2. Вопрос о последнем проекте, над которым работал кандидат.

3. Вопрос, не имеющий ответа.
4. Вопрос о программировании.
5. Вы удовлетворены?
6. У вас есть вопросы?

Я очень стараюсь не допускать, чтобы у меня сложилось предвзятое мнение о кандидате. Если человек кажется вам умным еще до того, как он вошел в комнату, лишь потому, что у него докторская степень из MIT, то что бы он ни говорил в течение часа интервью, не изменит вашего исходного мнения. Если вы считаете человека мужланом лишь потому, что он учился в местном колледже, то что бы он ни говорил, ваше первоначальное впечатление не изменится. Интервью похоже на крайне чувствительные весы: очень трудно оценить кого-либо на основе одночасового интервью, и такое предприятие может оказаться очень рискованным. Но если вы что-то узнали о кандидате заранее, то это знание подействует как гиря, брошенная на одну чашу весов, и интервью будет бесполезным. Однажды прямо перед интервью ко мне пришел агент по найму и сказал: «Вы просто *влюбитесь* в этого парня». Ну, как тут было не сойти с ума. На самом деле мне следовало сказать: «Если вы так уверены, что я полюблю его, так примите его на работу сами и не тратьте мое время на это интервью». Но я был молод и наивен и потому провел это интервью. Когда он говорил что-нибудь не очень умное, я думал про себя: ну, это исключение, лишь подтверждающее правило. На все, что он говорил, я смотрел сквозь розовые очки. В итоге я произнес «брать», хотя он был никудышным кандидатом. И что получилось? Все остальные, кто его интервьюировал, сказали *не брать*. Поэтому не слушайте агентов по найму, не расспрашивайте о человеке, пока не побеседовали с ним сами, и никогда не обсуждайте кандидата с другим интервьюирующим, пока вы оба не приняли самостоятельного решения. Таков научный метод.

### *1. Знакомство*

Этап знакомства на собеседовании должен помочь кандидату почувствовать себя свободно. Я спрашиваю, как он долетел. Секунд 30 я объясняю, кто я такой и как будет проходить собеседование. Я всегда предупреждаю кандидатов, что нас интересует, *как* они берутся за решение задачи, а не собственно ответ на вопрос.

### *2. Вопрос о последнем проекте*

Потом я задаю вопрос о каком-нибудь из последних проектов, над которыми работал кандидат. Беседуя со студентами, я спрашиваю их о дипломной работе, если они ее писали, или о каком-нибудь курсе, который они посещали, где был большой понравившийся им проект. Например, я могу спросить: «Какой курс вам больше всего понравился в последнем семестре? Он не обязательно должен быть связан с компьютерами.» Беседуя с опытными кандидатами, можно спросить, какое последнее задание было у них на предыдущей работе.

Задайте открытый вопрос, а сами расслабьтесь и слушайте, только вставляя иногда «расскажите об этом подробнее», если они запнутся.

На что надо обращать внимание, задавая открытые вопросы?

**Первое: Ищите увлеченность.** Умный человек увлекается проектом, над которым работает. Он очень возбуждается, рассказывая о предмете. Говорит быстро и оживленно. Активное *отрицательное* отношение может тоже служить хорошим признаком. «Мой последний начальник хотел делать все на компьютерах VAX, потому что в других он не разбирался. Какой болван!» Очень многие люди могут заниматься каким-то предметом, не испытывая к нему никакого личного отношения. Таких людей трудно чем-либо увлечь.

Плохие кандидаты просто равнодушны и не проявляют никакого энтузиазма во время интервью. Верное свидетельство страстного отношения кандидата к какому-то предмету – его способность забыть на какое-то время, что он находится на интервью. Иногда кандидаты ведут себя очень нервно, ощущая себя интервьюируемым; это естественно, и я не обращаю на это внимания. Но когда потом заводишь с ними разговор о Computational Monochromatic Art, они приходят в большое оживление и теряют все признаки нервозности. Это хорошо. Я люблю увлеченных людей. (Чтобы увидеть пример Computational Monochromatic Art, попробуйте выключить из розетки свой монитор.) Вы можете оспорить их слова (попробуйте: подождите, пока они произнесут что-то верное и скажите, что «этого не может быть»), и тогда они станут защищаться, даже если потели от волнения пять минут назад, потому что они так увлечены, что забывают о том, что вы вскоре примете решение, определяющее их жизнь.

**Второе: Хорошие кандидаты стараются понятно излагать вещи на любом уровне.** Я отказывал некоторым кандидатам, потому что, рассказывая о своем недавнем проекте, они были не в состоянии описать его язы-

ком, понятным нормальному человеку. Очень часто люди с техническим образованием полагают, что всем известна теорема Бейтса или аксиомы Пеано. Если они начинают говорить в таком духе, прервите их на минуту и скажите: «Не будете ли вы так любезны, просто в качестве упражнения, рассказать об этом так, чтобы было понятно моей бабушке». Часто после этого люди *все же* продолжают пользоваться техническим жаргоном, и понять их совершенно невозможно. *Гонг!* Вам не стоит брать их на работу, потому что у них не хватает ума, чтобы понять, как сделать свои мысли понятными другим.

**Третье: Если проект был групповым, посмотрите, есть ли у кандидата качества руководителя.** Он может рассказывать: «Мы работали над X, но начальник сказал Y, а клиент сказал Z». Я спрашиваю, как же *он* поступил. Хорошим ответом, скажем, будет: «Я собрал других членов бригады, и мы написали предложение...», а плохим: «Ну, я ничего *не мог* поделать, ситуация была безвыходной». Помните: *умен* и *доводит дело до конца*. Узнать о ком-либо, что он *доводит дело до конца*, можно, только проследив, была ли у него в прошлом тенденция доводить дело до конца. На самом деле можете даже прямо попросить их привести пример из недавнего прошлого, когда они взяли на себя руководство и завершили какое-то дело, например, преодолев инерцию своего учреждения.

### *3. Вопрос, не имеющий ответа*

Итак, третье место в моем плане интервью занимает *вопрос, на который нет ответа*. Это забавная часть. Идея в том, чтобы задать вопрос, на который невозможно ответить, и посмотреть, как возьмется за него кандидат. «Сколько оптометристов в Сиэтле?», «Сколько тонн весит памятник Вашингтону?», «Сколько заправочных станций в Лос-Анджелесе?», «Сколько настройщиков пианино в Нью-Йорке?».

Умные кандидаты понимают, что вы интересуетесь не их знаниями, и увлеченно стараются придумать какой-нибудь ответ. «Так, попробуем. В Лос-Анджелесе живет около 7 миллионов человек; у каждого человека в ЛА примерно 2,5 машины...» Конечно, ничего страшного, если они сильно ошибутся. Важно то, что они с энтузиазмом принялись за вопрос. Они могут попытаться определить производительность бензозаправочной станции. «Так, чтобы заправить бак, нужно 4 минуты, на станции примерно 10 насосов, и открыты они 18 часов в сутки...» Могут попытаться определить количество по площади. Иногда они проявляют удивительную изобре-

тательность или просят дать «желтые страницы» Лос-Анджелеса. Все это хорошие признаки.

Менее умные кандидаты волнуются и расстраиваются. Смотрят на вас, как на инопланетянина. Приходится руководить ими. «Представьте себе, что вы строите новый город размером с Лос-Анджелес. Сколько заправочных станций вы бы в нем открыли?» Можно давать маленькие подсказки. «Сколько времени нужно, чтобы заправить бак бензином?» И все равно приходится тащить таких неумных кандидатов, а они глупо сидят и ждут, когда вы придете к ним на помощь. Такие люди не способны решать задачи, и вам такие работники не нужны.

### *4. Вопрос по программированию*

В этой части интервью, которая должна быть самой продолжительной, я прошу кандидатов написать небольшую функцию на C (или на том языке, который им больше нравится). Вот типичные задачи, которые я задаю:

1. Инвертировать строку по месту.
2. Инвертировать связанный список.
3. Подсчитать количество единиц в байте.
4. Бинарный поиск.
5. Найти в строке самую длинную серию.
6. `atoi`.
7. `itoa` (замечательно, потому что им приходится использовать стек или `strrev`).

Не давайте задач, которые потребуют больше десятка строк кода: у вас не хватит на них времени.

Рассмотрим две задачи из этого списка детально.

1). Инвертировать строку по месту. Ни один из тех, кто когда-либо был у меня на собеседовании, не сделал этого правильно с первого раза. Все без исключения пытаются выделить новый буфер и записать инвертированную строку в этот буфер. Вопрос в том, кто будет выделять этот буфер. А кто освобождать его? Задавая этот вопрос десяткам кандидатов, я выяснил любопытный факт: большинство из тех, кто считает, что знает C, на самом деле не знают распределения памяти и указателей. Они просто не понимают этого. Поразительно, что такие люди вообще могут работать программистами. Этот вопрос позволяет оценить кандидата в разных отношениях.

Быстро ли работает их функция? Посмотрите, сколько раз они вызывают strlen. Я видел алгоритмы для strrev со сложностью $O(n^2)$, хотя она должна быть $O(n)$, потому что они многократно вызывают strlen в цикле.

Работают ли они с указателями? Это хороший признак. Многие «С-программисты» просто не понимают арифметики указателей. Да, я обычно не отклоняю кандидата из-за того, что он не владеет каким-то конкретным навыком. Однако я пришел к выводу, что понимание указателей в С – это не навык, а способность. На первом курсе обучения по компьютерной специальности в начале первого семестра всегда есть человек 200 ребят, писавших на BASIC сложные приключенческие игры для своих PC, когда им было по четыре года. Они успешно изучают в колледже старый добрый Pascal, пока однажды профессор не знакомит их с указателями, и вдруг *они перестают понимать*. Они просто больше ничего не понимают. Девяносто процентов учащихся сходит с дистанции и выбирает в качестве своей специализации политологию, а своим друзьям они объясняют, что в компьютерных классах было недостаточно привлекательных лиц противоположного пола, поэтому они переключились на другое. *По каким-то причинам большинство людей рождается без того участка мозга, который нужен для понимания указателей*. Для указателей требуется сложный вид дважды косвенного мышления, которое некоторым людям просто недоступно, но для хорошего программирования оно очень важно. Множество «скрипт-киддиз», начинавших программировать с копирования кусков JavaScript на свои веб-страницы и перешедших потом к изучению Visual Basic, не имеют понятия об указателях, и они не смогут создавать код того качества, которое вам нужно.

Посмотрим теперь на задание номер 3, подсчет битов в байте. Здесь нас не интересует, знают ли они поразрядные операции в С, потому что их можно найти в справочнике. Можете помочь им в этом. Интересно наблюдать, как они напишут функцию, которая подсчитывает все единицы в байте, а потом попросить их сделать ее более быстрой. Сообразительные кандидаты создадут справочную таблицу (в конце концов, в ней всего 256 элементов), и сделать это надо будет лишь один раз. С хорошими кандидатами можно завести интересный разговор о возможности разных компромиссов между памятью и скоростью. Попробуйте усложнить им задачу: скажите, что не хотите, чтобы тратилось время на создание справочной таблицы во время инициализации. Блестящие кандидаты могут предложить схему кэширования, в которой биты будут подсчитываться, когда встретятся в первый раз, и сохраняться в справочной таблице, чтобы не

считать их в следующий раз снова. Самые блестящие кандидаты пытаются придумать способ построить таблицу более эффективным способом, используя замеченные закономерности. Общая идея в том, чтобы они уточняли код, оптимизировали его и предлагали идеи для его улучшения.

Разъясните кандидатам, что вы понимаете, как тяжело писать код без редактора, и прощаете неаккуратность на бумаге. Вы также понимаете, что тяжело написать безошибочный код без компилятора, и учтете это.

Некоторые признаки хорошего программиста: у него обычно есть своя система выбора имен переменных, пусть даже простая. Хорошие программисты обычно используют очень короткие имена в переменных циклов. Если они дают переменной цикла имя вроде `CurrentPagePositionLoopCounter`, это верный признак того, что они писали мало кода в своей жизни. В C следите за хорошими привычками типа помещения константы слева от "==" (например, `if (0==x)` вместо `if (x==0)`), что предотвращает случайное использование "=" и последующее присваивание вместо проверки условия. Некоторые программисты считают такой прием неуклюжим и полагают, что такие ошибки должен находить компилятор, поэтому я не настаиваю, что нужно всегда писать `0==x`, но если вы увидите такую запись, то можете считать ее хорошим признаком. Хорошие программисты инстинктивно знают, какое условие надо написать в цикле `while`. Они не задумываясь, определяют, что для обхода массива в C надо писать `while (index < length)`, а не `<=`. В C++ они делают деструкторы виртуальными.

Хорошие программисты сначала составляют план, а потом пишут код, особенно когда в нем участвуют указатели. Например, если предложить им инвертировать связанный список, хорошие кандидаты всегда сначала нарисуют сбоку схему со всеми указателями и объектами, на которые они указывают. Они обязаны это сделать. Совершенно немыслимо, чтобы человек написал код для обращения связанного списка, не нарисовав сначала квадратики со стрелочками. Плохие программисты сразу начинают писать код.

### *5. Удовлетворены ли вы?*

Вы неизбежно обнаружите ошибку в их функции. Поэтому переходим к следующему пункту в моем плане интервью: вас удовлетворяет этот код? Вы можете спросить: «Хорошо, где в этом коде ошибка?» Основной дьявольский открытый вопрос. Все программисты допускают ошибки, и в этом нет ничего страшного; главное, чтобы они могли находить ошибки. В строковых функциях большинство студентов забывает завершить новую

строку нулем. Почти во всех функциях могут встретиться ошибки типа «плюс-минус один». Иногда забывают точку с запятой. Функция может некорректно обрабатывать строки нулевой длины или генерировать GPF при неуспехе malloc и т. д. Крайне редко вы встретите кандидата, который сразу напишет код без ошибок. В таких случаях этот вопрос еще интереснее. Когда вы говорите «в этом коде есть ошибка», они тщательно перечитывают код, и после этого вы можете посмотреть, насколько им удается быть дипломатичными и в то же время твердыми, утверждая, что их код безошибочен.

### *6. Есть ли у вас вопросы?*

В завершение интервью поинтересуйтесь, есть ли у кандидата вопросы. Помните, что хотя *вы* проводите интервью, но у хороших кандидатов есть большой выбор мест работы, и они в этот день решают для себя, хотят ли они работать у вас.

Некоторые интервьюирующие пытаются судить, насколько «интеллектуальные» вопросы задает кандидат. Лично мне все равно, какие вопросы они задают: к этому моменту я уже принял решение. Беда в том, что кандидату предстоит встречаться с пятью или шестью людьми в один день, и им трудно задавать пяти или шести людям различные и умные вопросы, поэтому если у них нет никаких вопросов, ничего страшного.

Я всегда оставляю в конце интервью минут пять для того, чтобы создать у кандидата хорошее представление о компании и работе. Это важно *даже в том случае, если вы приняли решение «не брать»*. Если вам повезло, и попался действительно хороший кандидат, надо сделать в этот момент все возможное, чтобы он захотел поступить к вам на работу. Но даже если кандидат плохой, надо, чтобы ему понравилась ваша компания, и он ушел с положительными впечатлениями.

Да, вспомнил: я обещал привести примеры действительно плохих вопросов, которых следует избегать.

## *О чем не спрашивать*

Прежде всего, не задавать вопросов, запрещенных законом. В США незаконно спрашивать обо всем, что касается расы, религии, пола, национальности, возраста, годности к военной службе, статуса ветерана, сексуальной ориентации или физических недостатков. Если в резюме сказано, что в 1990 году кандидат служил в армии, не спрашивайте, даже в порядке

приятной беседы, был ли человек в Кувейте. Это противозаконно. Если в резюме сказано, что кандидат учился в Технионе в Хайфе, не спрашивайте у него, израильтянин ли он, даже для поддержания беседы, потому что ваша жена израильтянка или вы любите фалафель. Это противозаконно.

Кроме того, избегайте вопросов, которые могут создать впечатление, что вы придаете статус решающих факторам, которые на самом деле таковыми не являются. Лучший пример такого рода – вопрос о наличии детей или брачном статусе. Он может создать впечатление, что вы считаете людей, у которых есть дети, уделяющими недостаточно времени работе, или боитесь, что они возьмут отпуск по беременности и рождению ребенка. Старайтесь задавать вопросы, которые относятся исключительно к работе, кандидатов на которую вы рассматриваете.

Наконец, избегайте вопросов-головоломок типа тех, где требуется сложить из шести спичек четыре одинаковых равносторонних треугольника. На большинство таких вопросов ответить невозможно, если не знать ответ заранее. Если человек говорит ответ, значит, он уже знал про эту головоломку раньше. Поэтому как интервьюирующий, вы не получите новой информации относительно «умен/доводит дело до конца», если будете смотреть, может ли кандидат сделать некоторый умственный кульбит.

Если в конце интервью вы убедились, что данный человек *умен* и *доводит дело до конца*, и еще четыре или пять интервьюирующих согласятся с вами, то, вероятно, вы не ошибетесь, взяв его на работу. Но если возникли какие-то сомнения, лучше подождать, пока не появится кто-нибудь лучше.

ГЛАВА ДВАДЦАТЬ ПЕРВАЯ

# Поощрительные выплаты – это зло

3 апреля 2000 года, понедельник

Майк Мюррей, бывший менеджер по кадрам Microsoft, сделал много промахов, но верхом их было введение награды «Ship It» за выпуск продукта вскоре после вступления его в должность. Идея была в награждении большой надгробной плитой из люцита размером с большой словарь в случае выпуска вашего продукта. Предполагалось, что это должно стимулировать вашу работу, потому что, видите ли, если вы не сделали свою работу, – никакого вам люцита! И как только Microsoft вообще что-то выпускала в *до*-люцитовую эпоху?

Программа «Ship It» была анонсирована с большой помпой на большом пикнике, устроенном компанией для своих служащих. За несколько недель до этого события на всей территории компании появились рекламные плакаты с изображением Билла Гейтса и не вполне понятной подписью «Чему он улыбается?». Может быть, он был рад, что у нас теперь есть стимул выпустить продукт вовремя? Со служащими обращались как с детьми, они это почувствовали и недвусмысленно выразили по этому поводу свое неодобрение. Бригада программистов Excel подняла огромный транспарант со словами «Почему бригаде Excel скучно?» Награду «Ship It» так презирали, что есть даже (невыдуманный) эпизод в классическом произведении Дугласа Коупленда (Douglas Coupland) «Microserfs»[1], в котором группа программистов пытается уничтожить люцитовую плашку с помощью паяльной лампы.

---

[1] Д. Коупленд «Рабы Майкрософта», АСТА, 2005. См. также на *www.wired.com/wired/archive/2.01/microserfs_pr.html*.

Пренебрежительное отношение к работникам, обладающим высочайшей квалификацией, не редкий случай. Почти во всех компаниях есть какая-нибудь оскорбительная и унизительная программа стимулов.

В двух компаниях, где я работал, самым напряженным временем года были полугодовые оценки продуктивности. По какой-то причине отделы кадров Juno и Microsoft заимствовали свои системы контроля продуктивности из одной и той же книги по менеджменту а-ля Дилберт, потому что обе были совершенно одинаковы. Сначала вы давали «анонимную» оценку своего непосредственного начальника (как будто это можно сделать честно). Затем вы заполняли необязательные формы «самооценки», которые ваш начальник «учитывал» при подготовке оценки вашей продуктивности. В итоге вы получали численную оценку по множеству неформальных категорий типа «хорошо сотрудничает с коллегами» в диапазоне от 1 до 5, где фактически можно было получить только оценки 3 или 4. Рекомендации о поощрении, представляемые менеджерами вверх по команде, совершенно игнорировались, и все получали премии практически на случайной основе. В этой системе никогда не учитывалось, что у людей есть разные и уникальные таланты и что для успешной работы бригады необходимы они все.

Оценки продуктивности вызывали напряжение по ряду причин. Многие мои друзья, особенно обладавшие яркими талантами, которые трудно измерить обычными мерками, часто получали плохие оценки продуктивности. Так, один из них обладал даром зажигать людей и энергично прокладывать курс, воодушевляя на движение вперед всех павших духом. Благодаря ему бригада была сплоченной. Но он обычно получал низкие оценки, потому что его руководитель не понимал той роли, которую он играл. Другой обладал выдающимся стратегическим умом; обсудив с ним подходы к той или иной проблеме, коллеги работали значительно лучше. Он больше других тратил времени на проверку новых технологий и в этом качестве был незаменим для остальной бригады. Но если измерять его результаты количеством строк кода, то они были ниже средних, а его менеджер был слишком глуп, чтобы понять его вклад в работу в других отношениях, поэтому он тоже всегда получал низкие оценки. Совершенно очевидно, что отрицательные оценки удручающим образом влияют на моральный дух. На самом деле даже позитивные оценки, но не настолько позитивные, как ожидает этот человек, тоже оказывают отрицательное воздействие на моральный дух.

Действие, оказываемое оценками на моральный дух, однобоко: отрицательные оценки сильно ему вредят, а положительные не влияют на моральный дух или продуктивность. Те, кто получает положительную оценку, уже и так работают продуктивно. Положительные оценки вызывают у них ощущение, что ради этого они и стараются, как собаки Павлова, работающие в надежде на поощрение, а не как профессионалы, которых действительно волнует качество делаемой ими работы.

И здесь есть загвоздка. Большинство людей считают, что они хорошо делают свою работу (даже если это не так). Это маленький обман, с помощью которого наше сознание пытается сделать нашу жизнь терпимой. Поэтому, когда каждый считает, что он работает хорошо, а оценки его работы всего лишь *объективны* (чего тоже трудно достичь), большинство *людей разочарованы своими оценками*. Какими потерями для морального духа это оборачивается, трудно даже представить. В тех бригадах, где оценки выставляются честно, их оглашение часто приводит к тому, что моральный дух оказывается подавлен на неделю, к плачу и иногда к увольнению. Они раскалывают бригаду, часто из-за зависти низко оцененных к высоко оцененным – процесс, который Демарко и Листер назвали *истреблением командного духа* (*teamicide*), непреднамеренным разрушением спаянных коллективов.[1]

Элфи Кон писал в ставшей классической статье в «Harvard Business Review»:

> ...не менее двух десятков исследований, проведенных за последние тридцать лет, убедительно показали, что те, кто рассчитывает получить вознаграждение за выполнение задания или успешную работу над ним, работают просто хуже тех, кто не рассчитывает получить никакой награды.[2]

Он делает вывод, что «системы морального или материального стимулирования работников неэффективны». Демарко и Листер идут еще дальше, прямо заявляя, что любые типы системы конкуренции среди работни-

---

1 Т. Демарко и Т. Листер «Человеческий фактор: успешные проекты и команды», 2-е издание, СПб.: Символ-Плюс, 2005.

2 Alfie Kohn (Элфи Кон) «Why Incentive Plans Cannot Work» (Почему бесполезны схемы стимулирования), *Harvard Business Review*, 1 сентября 1993г. См. *www.hbsp.harvard.edu/hbsp/prod_detail.asp?93506*.

ков, все типы наград и наказаний и даже старый способ «обнаружить, что кто-то делает что-то хорошо, и поощрить его», приносят больше вреда, чем пользы. Если вы даете кому-то *положительное* подкрепление (например, дурацкие церемонии награждения почетными значками в компаниях), то намекаете этим на то, что он старался только ради люцитовой плашки, что он не настолько независим, чтобы работать без расчета получить пирожок. Это оскорбительно и унизительно.

Большинству менеджеров в организациях, разрабатывающих ПО, не остается ничего иного, как согласиться с уже сложившейся там системой оценки производительности. Если вы оказались в такой ситуации, то единственный способ предотвратить истребление команды – дать каждому члену бригады преувеличенную оценку. Но если возможность выбора у вас есть, то советую избегать всеми силами разных оценок продуктивности, поощрительных премий и дурацких конкурсов на лучшего работника месяца.

ГЛАВА ДВАДЦАТЬ ВТОРАЯ

# Пять (неуважительных) причин, по которым у вас нет тестеров

30 АПРЕЛЯ 2000 ГОДА, ВОСКРЕСЕНЬЕ

В 1992 г. Джеймс Глейк (James Gleick) много писал о проблемах, связанных с ошибками в ПО. Глейк, хорошо известный журналист и эссеист, пишущий о науке и технологиях, высказался о вышедшей тогда новой версии Microsoft Word для Windows как об *ужасной*. Он написал пространную статью в воскресном выпуске *New York Times Magazine*, в которой подверг разработчиков Word беспощадной критике за невнимание к пожеланиям клиентов и выпуск продукта, нашпигованного ошибками.[1]

Затем, будучи клиентом местного интернет-провайдера Panix (и моего, как оказалось, провайдера тоже), он пожелал автоматически сортировать и фильтровать свою почту. В UNIX эта задача возлагается на утилиту procmail, которая работает весьма таинственным образом, предоставляя к тому же такой интерфейс, что *даже* самые верные фанаты UNIX признают его неясным.

Во всяком случае м-р Глейк сделал в procmail какую-то простую опечатку и потерял в результате всю свою почту. Придя в ярость, он решил создать собственную компанию, предоставляющую доступ в Интернет. Он нанял на работу программиста Юдея Айвейчури (Uday Ivatury) и создал Pipeline, которая действительно заметно опередила свое время: это был первый коммерческий провайдер доступа в Интернет, предоставлявший хоть какой-то графический интерфейс.

---

[1] James Gleick (Джеймс Глейк) «Chasing Bugs in the Electronic Village», *New York Times Magazine*, 4 августа 1992 г. См. также на *www.around.com/bugs.html*.

Конечно, у Pipeline были свои трудности. В самой первой версии был реализован протокол без коррекции ошибок, поэтому нередко сообщения искажались или происходила авария. Как и в любых программах, в ней были ошибки. В 1993 году я собирался поступить на работу в Pipeline. Во время собеседования я спросил м-ра Глейка, что он думает по поводу той статьи: «Вы оказались по другую сторону баррикад, теперь-то вы понимаете, как трудно создавать хорошие программы?»

Глейк упорствовал. Он отрицал, что в Pipeline есть ошибки. Он считал, что хуже Word нет ничего. Он сказал мне: «В один прекрасный день вы, Джоэл, тоже станете ненавидеть Microsoft». Меня несколько озадачило, что после года работы в качестве разработчика ПО, а не просто пользователя, он совершенно не почувствовал, как трудно создавать не содержащие ошибок и простые в обращении программы. В итоге я бежал, отказавшись от предложения работы. (Pipeline была куплена PSI, самым странным интернет-провайдером на свете, после чего ее без лишних церемоний прикончили.)

В программном обеспечении существуют ошибки, а процессоры чрезвычайно привередливы. Они напрочь *отказываются* работать, если задача не растолкована им до самой мелкой мелочи, и капризничают ну просто как *дети малые*. Когда я беру с собой лэптоп из дома, он часто отказывается работать, потому что не может найти сетевой принтер, к которому привык. *Ребенок да и только*. А дело-то все наверняка в какой-то единственной строчке кода, где есть крохотная, незаметная, почти несущественная ошибка.

Поэтому вам решительно и абсолютно необходим отдел, контролирующий качество, ОТК, так сказать. Обычно на одного тестера приходится два программиста (или больше, если программа должна работать в различных сложных конфигурациях или нескольких операционных системах). Каждый программист должен тесно сотрудничать с одним тестером и передавать ему свои личные сборки настолько часто, насколько это будет необходимо.

Этот отдел должен быть независимым и сильным и не должен подчиняться бригаде разработчиков, а его руководитель должен обладать правом вето на выпуск любого продукта, не прошедшего проверку.

Моя первая настоящая работа, связанная с программированием, была в Microsoft – компании, не очень *известной* высоким качеством своего кода, но которая тем не менее принимает на работу большое количество тес-

теров программного обеспечения. Поэтому я мог предполагать, что в каждом подразделении разработчиков ПО есть свои тестеры.

Действительно, во многих местах они есть. Но удивительно, что очень часто тестеров нет. И правда, многие команды не считают нужным иметь тестеров.

Казалось бы, после всей этой безумной борьбы за качество в 80-х годах, в связи с которой вспоминаются бессмысленные международные сертификаты «качества», подобные ISO-9000, и навязшие в зубах слова, типа «шесть сигм», сегодняшние менеджеры должны понимать, что выпуск высококачественных продуктов оправдан с точки зрения успешности бизнеса. Да, они понимают это. У многих это отложилось в голове. И все равно они находят массу причин, чтобы не держать тестеров, и все эти причины неуважительные.

Надеюсь, что мне удастся объяснить, почему их возражения несостоятельны. Если вы спешите, не читайте эту главу дальше, а просто пойдите и наймите на полный рабочий день одного тестера на каждого работающего у вас полный день программиста.

Вот самые распространенные заклинания, которыми чаще всего пытаются оправдать отсутствие тестеров.

## *1. Ошибки – результат лени программистов*

Некоторые рассуждают: «Если мы наймем тестеров, то программисты станут небрежными и будут писать код с ошибками. А если тестеров не будет, то мы сможем заставить программистов писать правильный код сразу.»

Черта с два. Если вы так считаете, то либо никогда сами не писали код, либо очень нечестно рассуждаете о том, как пишется код. Ошибки по определению случаются потому, что программисты *не видят* их в собственном коде. Очень часто для обнаружения ошибки достаточно, чтобы на код взглянул кто-то другой.

Когда я работал в Juno, то обычно писал код в одном и том же стиле в соответствии со своими привычками и много пользовался мышью. У нашего очаровательного и сверхквалифицированного тестера были несколько иные привычки: она чаще работала с клавиатурой (и фактически строго тестировала интерфейс, вводя самые разнообразные комбинации входных данных). В результате быстро выявилась целая *уйма* ошибок. Иногда она сообщала мне, что интерфейс вообще не работает, *на все 100%*, хотя у меня

он всегда работал. Глядя, как она воспроизводит ошибку, я рвал на себе волосы. Alt! Она нажала клавишу Alt! Почему я этого не проверил?

## *2. Моя программа лежит в Интернете. Если понадобится, я исправлю ошибку за секунду*

Ха-ха-ха-ха! Верно, Интернет позволяет распространять исправления ошибок гораздо быстрее, чем в прежние времена коробочных продуктов. Но нельзя недооценивать стоимость исправления ошибки, даже через веб-сайт, после того как работа над проектом уже прекращена. Во-первых, исправляя одну ошибку, можно внести в программу новые. Но хуже то, что при ближайшем рассмотрении процедура выкладывания новых версий в Интернет оказывается весьма дорогостоящей. Кроме того, это производит неблагоприятное впечатление на клиентов, что подводит нас к неуклюжему возражению номер 3.

## *3. Пусть вместо меня программу тестируют мои клиенты*

Вспомним пресловутую и нашумевшую «защиту Netscape». Эти бедолаги нанесли невероятный ущерб своей репутации благодаря принятой ими методологии «тестирования»:

1. Когда программисты закончили работу примерно наполовину, выкладывать ПО в Интернет без всякого тестирования.
2. Когда программисты *говорят*, что они готовы выкладывать ПО в Интернет без всякого тестирования.
3. Повторить первые два шага шесть или семь раз.
4. Назвать одну из этих версий окончательной.
5. Выпускать версии .01, .02, .03 каждый раз, когда в CNET сообщается о неприятной ошибке.

Эта компания была пионером идеи «широкого бета-тестирования». Буквально *миллионы* людей загружали эти недоделанные и полные ошибок версии. Первые несколько лет почти у всех, кто пользовался Netscape, стоял какой-нибудь предрелиз или бета-версия. В результате широко распространено мнение, что в программах Netscape очень много ошибок. Даже если окончательная версия была в приемлемой мере отлажена, Netscape напугала настолько *много* людей своими неотлаженными версиями, что

*в среднем* у большинства сложилось впечатление о плохом качестве их продукции.

Кроме того, вся идея «отладки клиентами» состоит в том, что они находят ошибки, а вы их устраняете. К сожалению, ни Netscape, ни какая-либо другая компания в мире не в состоянии просмотреть сообщения об ошибках от 2 миллионов клиентов и определить, какие из них существенны. Когда я хотел сообщить об ошибках в Netscape 2.0, сайт, принимавший сообщения, регулярно падал, и я просто не смог передать отчет (который все равно пропал бы, как в черной дыре). Но в Netscape ничему не научились. Тестеры текущей «предварительной» версии 6.0 жалуются в телеконференциях, что веб-сайт по-прежнему не позволяет передать сообщение. Годы прошли, а проблемы остались прежними!

Могу поспорить, что среди всех этих бесчисленных сообщений об ошибках есть группа из пяти или десяти действительно *очевидных* ошибок. И одна или две трудно обнаружимых ошибки, о которых кто-то не поленился сообщить и которые потерялись теперь, как иголки в стоге сена, потому что никто не будет просматривать все эти сообщения, и они пройдут незамеченными.

Худшее в таком виде тестирования – крайне неприятное впечатление, создающееся о компании. Когда UserLand выпустила первую версию своего флагманского продукта Frontier для Windows, я загрузил его и стал читать руководство. К сожалению, Frontier несколько раз аварийно завершилась. Я буквально выполнял инструкции, в точности следуя тому, что напечатано в руководстве, и не мог добиться, чтобы программа проработала дольше двух минут. Я пришел к выводу, что никто в UserLand не провел даже *минимального* тестирования, чтобы убедиться, что *руководство* актуально. Плохое впечатление о качестве продукта очень надолго отвратило меня от Frontier.

## *4. Все, у кого достаточно квалификации, чтобы быть хорошим тестером, не хотят заниматься тестированием*

С этим трудно спорить. Найти хороших тестеров очень трудно. С ними, как с программистами: лучшие из них *на порядок* выше средних. В Juno одним из тестеров была Джилл Макфарлейн, которая находила *втрое больше ошибок, чем другие четверо тестеров, вместе взятых*. Я не преувеличиваю; я действительно измерял. Она была в *двенадцать раз*

продуктивнее среднего тестера. Когда она ушла, я послал директору письмо, в котором написал, что предпочел бы видеть Джилл по понедельникам и вторникам, чем всех остальных ежедневно.

К сожалению, большинство толковых людей устает от ежедневного тестирования, поэтому лучшие тестеры держатся примерно три-четыре месяца, а потом уходят.

С этой проблемой нельзя ничего сделать, кроме как признать ее существование и пытаться ее учитывать. Можно предложить такие варианты:

- Рассматривать тестирование как повышение по сравнению с технической поддержкой. Как ни скучно заниматься тестированием, но это все же лучше, чем общаться по телефону с разгневанными пользователями, и таким способом можно бороться с некоторой текучестью в службе технической поддержки.
- Дать тестерам возможность продолжить карьеру, разрешив посещение курсов программирования, и побуждать наиболее способных из них разрабатывать автоматизированные средства тестирования с применением программных средств и языков сценариев. Это куда как интереснее, чем снова и снова тестировать одно и то же окно диалога.
- Смириться с тем, что среди лучших тестеров будет высокая текучесть. Активно заниматься набором сотрудников, чтобы обеспечить постоянный приток кадров. Не прекращать поиск сотрудников только потому, что в какой-то момент у вас заполнен штат. Золотой век не вечен.
- Ищите «нетрадиционных» сотрудников: толковых подростков, студентов, подрабатывающих пенсионеров. Можно создать прекрасный отдел тестирования, если взять двух-трех первоклассных работников на полный рабочий день и нанять молодых людей из Bronx Science (отличный колледж в Нью-Йорке), зарабатывающих летом на оплату учебы.
- Нанимайте временных работников. Наймите десяток временных работников, которые несколько дней помучают вашу программу, и вы увидите, какую кучу ошибок они найдут. Двое или трое из них окажутся умелыми тестерами, и тогда стоит оформить их на полный рабочий день. Учтите заранее, что от некоторых из этих временщиков толка не будет; отправьте их домой и не печальтесь. Для того и существуют агентства по временному найму.

А вот этого делать *не следует*:

- Не вздумайте предлагать у себя работу выпускникам колледжей по специальности «информатика», предупреждая, что «каждый должен поработать тестером, а потом будет переведен в программисты». Я насмотрелся таких случаев. Из программистов не получается хороших тестеров, и вы потеряете хорошего программиста, которого труднее найти.

И наконец, главная неуважительная причина, по которой не берут на работу тестеров:

## *5. Я не могу себе этого позволить!*

Эта отговорка самая глупая, и ее легче всего развенчать.

Как бы ни трудно было найти тестеров, они *все же* дешевле, чем программисты. Гораздо дешевле. А если вы не наймете тестеров, вам придется заставить программистов заниматься тестированием. А если вам не нравится, что тестеры часто меняются, посмотрим, как вам понравится, если придется искать замену классному программисту, получающему 100 000 в год, которому надоело ваше требование «потратить несколько недель на тестирование перед выпуском нашего продукта» и который перешел в другую, более профессиональную компанию. Вы могли бы нанять трех тестеров на *год* за гонорар агента по найму, который найдет вам программиста на замену.

Скупость на тестеров оказывается такой ложной экономией, что я просто поражаюсь, сколь многие люди этого не понимают.

ГЛАВА ДВАДЦАТЬ ТРЕТЬЯ

# Многозадачность придумана не для разработчиков

12 ФЕВРАЛЯ 2001 ГОДА, ПОНЕДЕЛЬНИК

Когда руководишь группой программистов, очень важно научиться правильно распределять между ними задачи. Говоря просто, *надо каждому найти занятие*. В иврите есть выражение, примерно означающее «раскидать папки» (потому что каждому вы бросаете на колени его папку). А от вашего решения – кому и что отдать – очень сильно зависит конечная продуктивность. Ошибетесь – и получите трудную ситуацию, когда никто ничего не доводит до конца и все ворчат, что «тут ничего не делается».

Эта книга рассчитана на программистов, поэтому я хочу дать им для разминки такую задачу.

Допустим, есть две вычислительные задачи, A и B. Для выполнения каждой требуется 10 секунд процессорного времени. В вашем распоряжении один процессор, других заданий (предположим для простоты) в очереди нет.

Процессор позволяет организовать многозадачность. Это означает, что можно выполнить вычисления одно за другим, задав последовательную обработку:

| Вычисление A | | | | | | | | | | Вычисление B | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |

Или организовать многозадачность:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

В режиме многозадачности на данном конкретном ЦП на каждую задачу выделяется по 1 секунде, а переключение задач происходит мгновенно.

Что бы вы предпочли? Большинство людей интуитивно полагает, что многозадачность лучше. В обоих случаях надо ждать 20 секунд, чтобы получить оба результата. Но посмотрите, сколько времени пройдет до получения результатов *каждого* из вычислений.

В обоих случаях результаты вычисления B (показано черным цветом) поступят через 20 секунд. А вычисления A? При многозадачной работе его результаты будут известны через 19 секунд, а при последовательной обработке... через 10.

Иными словами, в этом приятном вымышленном примере *среднее время вычисления* меньше (15 секунд вместо 19,5) для последовательной обработки, а не параллельной. (На самом деле пример не такой уж искусственный, в его основе реальная задача, которую пришлось решать Джареду.)

| Метод | Вычисление A, сек. | Вычисление B, сек. | В среднем, сек. |
|---|---|---|---|
| Последовательный | 10 | 20 | 15 |
| Параллельный | 19 | 20 | 19,5 |

Выше я сказал, что переключение между задачами происходит мгновенно. Но на реальных процессорах переключение задач отнимает некоторое время – столько, сколько нужно, чтобы сохранить состояние регистров ЦП для одной задачи и загрузить состояние регистров для другой. На практике оно настолько мало, что им можно пренебречь. Но чтобы сделать себе жизнь интереснее, допустим, что переключение задач длится полсекунды. Тогда положение становится еще хуже:

| Метод | Вычисление A | Вычисление B | В среднем |
|---|---|---|---|
| Последовательный | 10 сек | 20 +1 переключение = 20,5 сек | 15,25 сек |
| Параллельный | 19 +18 переключений = 28 сек | 20 +19 переключений = 29,5 сек | 28,75 сек |

А теперь (я знаю, что это чепуха, но хочу доставить себе удовольствие) предположим, что переключение задач длится минуту.

| Метод | Вычисление A | Вычисление B | В среднем |
|---|---|---|---|
| Последовательный | 10 сек | 20 +1 переключение = 80 сек | 45 сек |
| Параллельный | 19 +18 переключений = 1099 сек | 20 +19 переключений = 1160 сек | почти 19 минут! |

Чем дольше переключаются задачи, тем невыгоднее параллельность.

Понятно, что само по себе это весьма банально. Скоро какие-нибудь недоумки завалят меня гневными письмами, крича, что я «против» многозадачности: «Вы что, хотите вернуться во времена DOS, когда надо было выйти из WordPerfect, чтобы запустить 1–2–3?»

Нет, я не это имею в виду. Я просто хочу, чтобы вы согласились, что в данном примере:

1. Последовательная обработка *в среднем* дает результаты быстрее.
2. Чем дольше переключаются задачи, тем дороже приходится платить за многозадачность.

Но вернемся к управлению людьми; это интереснее, чем управлять ЦПУ. Фокус здесь в том, что, когда руководишь именно *программистами*, переключение задач действительно занимает очень и очень большое время.[1] Дело в том, что программисту приходится держать в голове массу вещей одновременно. И чем больше он помнит, тем он эффективнее. Когда программист работает в полную силу, у него в голове одновременно хранится бесчисленное количество вещей – все, включая имена переменных, структуры данных, важные API, названия вспомогательных функций, которые он для себя написал и часто вызывает, даже имя каталога, в котором он хранит свой код. Если отправить этого программиста недели на три в отпуск на Крит, он забудет *все* начисто. Человеческий мозг кажется устроенным так, будто из оперативной памяти выгружает все на магнитную ленту, откуда потом восстанавливает данные целую вечность.

---

[1] Описываемые в этой главе эффекты, связанные с переключением внимания разработчиков между различными задачами, чем так часто злоупотребляют администраторы проектов, – это, безусловно, большое зло. Но эффекты эти есть следствие глубинных и тонких психологических процессов. Все же приводимые автором «аналогии» из многопоточной обработки ошибочны. В первом примере автор просто «жонглирует» понятием «среднее время» и использует его не по назначению. Со вторым же примером еще хуже; здесь автор просто ошибается, попадая в ловушку многократно повторяемого заблуждения, что переключение между процессами A и B отбирает много производительности. Много по сравнению с чем? При этом он упускает из виду, что в многозадачной ОС переключение между временными «порциями» одной задачи (переключение с задачи A на задачу A) осуществляется теми же механизмами ОС, с сохранением-восстановлением контекста, и очень слабо зависит от того, происходит ли переключение единственной задачи, двух потоков A и B, принадлежащих единому процессу, или двух независимых процессов A и B. Если уж вы используете многозадачную ОС, то это безальтернативная плата за удовольствие! – *Примеч. науч. ред.*

А как долго на самом деле? Вот пример: мы недавно бросили все дела (разработку нашего продукта CityDesk) и три недели кряду помогали клиенту, у которого возникла чрезвычайная ситуация. Когда мы вернулись к себе в офис, прошло, пожалуй, *еще* недели три, прежде чем мы достигли прежней максимальной скорости в работе над CityDesk.

Вы никогда не замечали, что если дать человеку одно задание, то он прекрасно с ним справится, а если дать *два*, то от него уже не дождешься никаких толковых результатов? Либо он хорошо выполнит одну работу с полным пренебрежением к другой, либо обе работы будет делать медленно, как улитка. Дело в том, что в программировании переключение задач происходит очень долго. Когда мне приходится вести одновременно два программных проекта, время переключения задач составляет, по моим ощущениям, часов шесть. При восьмичасовом рабочем дне это приводит к тому, что в режиме многозадачности моя продуктивность ограничивается двумя часами в день. Довольно плачевный результат.

Поэтому получается, что, когда даешь кому-нибудь сразу два задания, нужно еще радоваться, если он «забросит» одно из них и будет заниматься только другим, потому что в результате он сделает больше и в среднем закончит задачи быстрее. Действительный урок, который следует из этого извлечь: *не разрешайте сотрудникам заниматься одновременно более чем одной задачей*. И следите, чтобы они знали, что это за задача. Хорошие администраторы считают своей обязанностью *устранять препятствия* к тому, чтобы люди могли сосредоточиться на *одной задаче* и довести ее до конца. Если возникает что-то чрезвычайное, постарайтесь справиться с этим самостоятельно, не перекладывая на программиста, глубоко погрузившегося в работу над проектом.

ГЛАВА ДВАДЦАТЬ ЧЕТВЕРТАЯ

# То, чего делать нельзя, часть первая[1]

6 апреля 2000 года, четверг

Наконец-то вышла первая общедоступная бета-версия Netscape 6.0. Версии 5.0 так и не было. Последней основной версией была 4.0, выпущенная в 1997 г. Для Интернета три года – это *очень* много. Все это время Netscape беспомощно наблюдала, как стремительно сокращается ее доля рынка.

Я немного льщу им, критикуя за слишком длинный перерыв между выпусками очередных версий. А, так они *специально* это сделали, так получается?

Ну да. Именно так. Специально, причем совершили *самую тяжелую стратегическую ошибку*, которую может сделать компания-разработчик программного обеспечения:

**Они решили переписать весь код заново.**

Netscape не была первой компанией, совершившей такую ошибку. То же самое сделала Borland, когда купила Arago и попыталась превратить его в dBase для Windows – обреченный проект, с которым они тянули столько, что их кусок пирога достался Microsoft Access. Ту же ошибку они повторили, переписав заново Quattro Pro и вызвав у всех удивление скудостью возможностей того, что получилось в результате. Microsoft тоже едва не совершила такую же ошибку, попытавшись переписать Word для Windows с чистого листа в обреченном проекте под названием Pyramid, который

[1] Других частей не будет. Заглавие просто обыгрывает фильм Мела Брукса «История мира: часть I», не имеющий никакого отношения к программированию или данному очерку.

был прекращен, отменен и забыт. К счастью для Microsoft, она не прекращала работу над прежним кодом, поэтому кое-что им удалось выставить на рынок и потерпеть в итоге убытки, но не стратегическую катастрофу.

Мы – программисты. Все программисты в глубине души – архитекторы, а первое, что хочет сделать архитектор, прибыв на строительный участок, – это выровнять его бульдозером и построить нечто грандиозное. Нас не воодушевляет частичная реконструкция: копаться, улучшать, разбивать цветочные клумбы.

Есть скрытая причина, по которой программисты всегда хотят выкинуть старый код и начать все сначала. Это происходит потому, что прежний код кажется им запутанным.[1] Я так думаю, что *они, вероятно, ошибаются*. Старый код кажется запутанным, потому что есть важный фундаментальный закон программирования:

**Читать код труднее, чем писать его.**

Из-за этого так трудно повторно использовать код. Из-за этого у каждого в вашей бригаде есть собственная функция, с помощью которой он разбивает строку на массив строк. Свою функцию пишут потому, что это проще и интереснее, чем разбираться, как работает старая.

Следствие из этой аксиомы вы можете увидеть, если спросите практически у любого программиста, что это за код, над которым он сейчас работает. «Абсолютная неразбериха, – ответит он вам. – Ни о чем так не мечтаю, как выкинуть его и начать все сначала».

В чем же неразбериха?

«Да посмотрите на эту функцию, – говорят они. – Целых две страницы! Зачем здесь весь этот код? Я не знаю, для чего нужна половина этих вызовов API.»

Перед тем как Borland выпустила новую электронную таблицу для Windows, Филип Кан, основатель Borland и интересная личность, неоднократно заявлял в прессе, что Quattro Pro будет гораздо лучше Microsoft Excel, потому что написана с чистого листа. Полностью новый исходный код! Как будто исходный код портится от *ржавчины*.

---

[1] Здесь с автором нельзя не согласиться; более того, сказанное следовало бы даже усилить: наблюдения за многочисленными программистами приводят к мысли, что сила желания «переписать все с нуля» обратно пропорциональна опыту и истинной квалификации программиста. Только с опытом программист начинает с вниманием относиться к чужому коду и чувствует в себе силы взяться за его модернизацию. – *Примеч. науч. ред.*

Мысль о том, что новый код лучше, чем старый, явно абсурдна. Старым кодом уже *пользовались*. Он *проверен*. Найдено и *исправлено* множество ошибок. Что в нем плохого? В нем не заведутся ошибки лишь оттого, что он лежит у вас на жестком диске. *Au contraire*, малыш! Разве программное обеспечение похоже на старый Dodge Dart, ржавеющий в гараже без движения? Или на плюшевого медведя, который, как бы это сказать, вульгарен, если он не *весь с ног до головы новый*?

Опять же об этой функции на две страницы. Да, я знаю, что эта функция просто должна выводить окно, но почему-то она обросла неизвестно чем? А я вам скажу почему: это исправления ошибок. Одна из них исправляет ошибку, которая обнаружилась у Нэнси, когда она устанавливала программу на компьютере без Internet Explorer. Другая исправляет ошибку, которая возникает при нехватке памяти. Еще одна исправляет ошибку, которая возникала, когда файл был на дискете, а пользователь выдергивал ее во время работы. Этот вызов `LoadLibrary` некрасив, но благодаря ему код работает под старыми версиями Windows 95.

Каждая из этих ошибок обнаружилась только по прошествии недель практической эксплуатации программы. Возможно, и программист еще потратил пару дней на то, чтобы воспроизвести ошибку на стенде и исправить ее. Для большинства ошибок заплатка может состоять из одной строки кода, а то и пары символов, но эта пара символов стоила изрядного количества труда и времени.

Выкидывая код и начиная все с начала, вы выбрасываете и все эти знания. Все эти накопленные исправления ошибок. Годы программистского труда.

Вы выбрасываете свое лидирующее положение на рынке. Вы дарите два или три года своим конкурентам, а, поверьте мне, в индустрии ПО это *очень много*.

Вы ставите себя в очень опасное положение, теперь вам придется несколько лет предлагать старую версию своего кода, совершенно не имея возможности вносить в него стратегически важные изменения или выпустить новую функцию, затребованную рынком, потому что у вас нет готового для поставки кода. С таким же успехом можно было вообще закрыться на это время.

Вы тратите уйму денег на то, чтобы написать код, который уже существует.

Есть ли другие варианты? По общему согласию прежний код Netscape был действительно плох. Что ж, может быть, он и *был плох*, но... он очень неплохо работал на огромном количестве компьютерных систем.

Когда программисты говорят, что код жутко запутан (а они всегда так говорят), то это значит, что в нем есть проблемы трех видов.

Во-первых, проблемы архитектуры. Код плохо разделен на отдельные компоненты. Та часть, которая осуществляет работу в сети, вдруг начинает выводить собственные диалоговые окна, хотя это должен делать код интерфейса пользователя. Эти проблемы можно решать, постепенно и осторожно перемещая код, осуществляя рефакторинг, модифицируя интерфейсы. Это может делать один программист, который будет аккуратно работать и вносить свои изменения сразу все целиком, чтобы не нарушать работу остальных. Даже довольно серьезные изменения архитектуры можно проводить, *не выкидывая весь код*. В проекте Juno мы однажды несколько месяцев занимались модификацией архитектуры: просто перемещали части кода, причесывали их, создавали разумные базовые классы и элегантные интерфейсы между модулями. Но делали все это осторожно, на основе имеющегося кода, не внося новых ошибок и не выбрасывая действующий код.

Вторая причина, по которой программисты считают свой код запутанным, это его неэффективность. Говорили, что код рендеринга (перерисовки) в Netscape работал медленно. Но это же лишь малая часть проекта, которую можно оптимизировать или даже переписать. Зачем переписывать все? Если надо оптимизировать скорость, то 1% работы обеспечивает 99% результата.[1]

В-третьих, код может быть чертовски уродлив. Я работал над одним проектом, в котором был тип данных под именем `FuckedString`. В другом проекте сначала действовало соглашение помечать члены-переменные символом подчеркивания в начале имени, но потом они перешли на более общепринятое `"m_"`.[2] В итоге половина функций начиналась с `"_"`, а другая половина – с `"m_"`, что смотрелось ужасно. Честно говоря, подобные «проблемы» решаются за пять минут с помощью макроса в Emacs, и не нужно начинать все сначала.

---

[1] Это достаточно вольный пересказ известного закона «пяти процентов»: «...за 95% достижимого в идеале ускорения работы программного кода отвечают 5% операторов самого кода». – *Примеч. науч. ред.*

[2] Это и есть упоминавшаяся ранее в тексте «венгерская нотация», широко принятая в фирме MicroSoft. – *Примеч. науч. ред.*

Важно помнить, что если вы начинаете писать код с нуля, то *нет абсолютно никаких оснований* полагать, что на этот раз у вас получится лучше, чем в предыдущий. Во-первых, у вас, наверняка, уже и той команды нет, которая работала над версией 1.0, так что едва ли у вас «стало больше опыта». Вы просто повторите большинство прежних ошибок и внесете новые, которых не было в первоначальной версии.

Старая мантра «будьте готовы к тому, что какую-то версию придется выкинуть» опасна, когда ее относят к крупным коммерческим приложениям. Если вы пишете экспериментальный код, то можно порвать на кусочки функцию, написанную на прошлой неделе, если вам пришел в голову лучший алгоритм. Это прекрасно. Можете реорганизовать класс, чтобы проще было им пользоваться. Это тоже прекрасно. Но выброс целой программы – опасное безумие, и если бы в Netscape были зрелые руководители с опытом работы в программной индустрии, возможно, они не наказали бы сами себя так жестоко.

## *Постскриптум*

Группа артистов, прежде известная под именем Netscape, в конце концов выпустила нечто под названием Mozilla – свой потрясающий полностью переработанный код. За то время, что им для этого потребовалось, они практически полностью утратили свою долю рынка.[1]

[1] Справедливости ради надо отметить, что за время, прошедшее после выхода книги, популярность Mozilla медленно, но неуклонно растет. В первую очередь этот прирост вызван тем, что Mozilla – это многоплатформенный продукт. И если в мире Windows его популярность растет медленно, то во многих других ОС (Linux, QNX) Mozilla становится чуть ли не основным броузером. – *Примеч. науч. ред.*

ГЛАВА ДВАДЦАТЬ ПЯТАЯ

# Секрет айсберга

13 февраля 2002 года

«Я не понимаю, что случилось с моими разработчиками, – размышляет главный начальник. – Проект начался отлично. Первую пару недель они работали, как сумасшедшие, и сделали отличный действующий прототип. Но с тех пор они топчутся на месте. Они просто не хотят больше работать». Он выбирает титановую клюшку Callaway и посылает помощника за ледяным лимонадом. «Вот уволю пару из этих тюфяков, тогда они зашевелятся!»

Тем временем разработчики, естественно, не имеют *ни малейшего понятия* о том, что что-то не так. На самом деле ничего не случилось. Они не отстают от графика.

Пусть с вами этого никогда не произойдет! Я расскажу вам, в чем секрет таких менеджеров, не имеющих технической подготовки, и ваша жизнь станет в тысячу раз легче. Это совсем просто. Посвящение в эту тайну сделает ваше общение с ними безоблачным (если только вы не станете обсуждать с ними коэффициент восстановления в их гольф-клубе).

Совершенно очевидно, что программисты и MBA говорят на разных языках. Я давно задумывался о проблемах общения в руководстве программными разработками, потому что мне достаточно ясно, что власть и вознаграждение достаются тем редким людям, которые умеют переводить с программистского на администраторский.

Почти все программное обеспечение, с которым я сталкивался за время работы в индустрии ПО, можно было бы назвать «спекулятивным». Имеется в виду, что оно создавалось не для конкретного клиента, а в расчете на то, что его купят *миллионы* людей. Но многим разработчикам ПО такая

роскошь недоступна. Они могут быть консультантами-разработчиками проекта для единственного клиента или штатными программистами, работающими над какой-нибудь сложной примочкой для бухгалтерской системы (или чем там еще могут заниматься внутрифирменные программисты – это для меня в известном смысле загадка).

Вы не замечали, работая над такими специальными проектами, что чаще всего причину задержек, неудач и общего несчастья можно, по существу, свести к тому, что «этот (здесь место для бранного эпитета) клиент сам не знал, что ему надо»?

Вот три разновидности такой патологии:

1. «Чертов заказчик непрерывно менял свои решения. Сначала ему была нужна архитектура клиент/сервер. Потом в самолете Delta Airlines ему попался журнал, где рассказывалось об XML, и он решил, что ему необходим XML. Теперь мы переписываем все так, чтобы оно действовало на основе парка маленьких роботов Lego Mindstorms».
2. «Мы сделали все *в точности, как они хотели*. В контракте все было расписано до мельчайших деталей. Мы создали именно то, что было оговорено в контракте. Теперь, когда мы поставили им продукт, они нами недовольны».
3. «Наш торговый представитель, чтоб ему пусто было, заключил с ними *контракт с фиксированной ценой* на разработку без четкой спецификации, а их юристы сумели вставить в контракт условие, что оплата происходит только после «приемки продукта заказчиком», в результате чего девять человек работали над их проектом два года, и им заплатили 800 долларов».

Если бы каждому молодому консультанту можно было загнать что-то в голову, просверлив ее сверхмощной дрелью со скоростью 2500 об/мин, то это было бы такое правило:

**Заказчики не знают, чего они хотят.**
**Не надейтесь, что заказчики будут знать, чего они хотят.**

Этого никогда не будет. Забудьте об этом.

Лучше исходить из того, что вам *все равно* придется что-то написать, и это что-то должно понравиться заказчику, но он будет несколько удивлен. *Вы* должны провести исследование. *Вы* должны придумать, как решить проблемы заказчика удобным для него образом.

Поставьте себя на место заказчика. Представьте себе, что вы только что заработали 100 000 000 долларов, продав Yahoo! свою компанию, и решили, что теперь можно сделать ремонт на кухне. Вы нанимаете знающего архитектора и объясняете ему, что кухня должна быть «такой же крутой, как у Уилла и Грэйс». Как это сделать, вы понятия не имеете. Вы не знаете, что вам нужна плита Viking и холодильник Subzero – таких слов просто нет в вашем лексиконе. Вы хотите, чтобы архитектор сделал все хорошо – а для чего же еще его наняли?

Апологеты экстремального программирования объяснят вам, что клиента надо пригласить *в помещение* и привлечь его к процессу проектирования на всех стадиях как члена бригады разработчиков. Я думаю, что это уж *слишком* «экстремально». Это что же, мой архитектор заставил бы меня присутствовать при проектировании моей кухни и требовал бы моего мнения по всем, даже мелким деталям? Мне это неинтересно, а если бы я хотел быть архитектором, то я бы им и был.

Нет, правда, вам *надо*, чтобы заказчик входил в состав вашей бригады? Тот, кого заказчик откомандирует к вам, окажется каким-нибудь беднягой из бухгалтерии, которого отправили к программистам, потому что он у них тормозит сильнее всех и его отсутствие будет почти незаметно. И будете вы, пока не закончите проектирование, изъясняться односложными словами.

Пусть ваш заказчик действительно не знает, что ему нужно. Сделайте проект сами, исходя из своего представления о предметной области. Пусть вам потребуется некоторое время, чтобы расширить свои представления об этой предметной области, или помощь эксперта в ней – это нормально, но проектирование программного обеспечения – это ваша работа. Если вы разберетесь с предметной областью и создадите хороший интерфейс пользователя, заказчик будет доволен.

Я обещал поделиться с вами секретом перевода с языка клиента (или технически безграмотного менеджера) на язык, который понимают программисты.

Вы знаете, что айсберг на 90% скрыт под водой? Так вот, программное обеспечение обычно на него похоже: в нем есть милый интерфейс пользователя, который требует процентов 10%, а 90% программистского труда скрыто от глаз. А если принять во внимание, что примерно половина времени уходит на отладку, то интерфейс требует лишь около 5% всех трудовых затрат. А если ограничить себя только *видимой* частью интерфейса, пикселами, тем, что вы увидите в PowerPoint, то речь пойдет вообще об 1% и менее.

Но секрет не в этом. Секрет в том, что *люди, не являющиеся программистами, этого не понимают.*

Существует ряд очень важных следствий из секрета айсберга.

## *Важное следствие номер один*

Если вы покажете непрограммисту экран с интерфейсом, который будет выглядеть на 90% хуже, он решит, что программа на 90% хуже.

Я выяснил это, работая консультантом, когда сделал демонстрацию крупного веб-проекта для группы руководящих сотрудников заказчика. Код проекта был завершен почти на 100%. Мы ждали только графического дизайнера, который подберет шрифты, цвета и нарисует элегантные 3-мерные закладки. Мы же тем временем использовали обычные шрифты и черно-белую графику, на экране было много отвратительных участков, и в целом все выглядело не слишком красиво. Но функциональность была реализована полностью и работала очень неплохо.

Что произошло во время демонстрации? В течение *всего совещания* клиенты лишь выражали недовольство графическим видом экрана. Они даже интерфейс не обсуждали. Только графический стиль. «*Вид неприятный*», – жаловался их руководитель проекта. Это все, о чем они были способны думать. Перевести их на обсуждение функциональности мы не смогли. Понятно, что работа над графическим дизайном отняла около одного дня. Такое впечатление, что они считали, будто пригласили на работу *маляров*.

## *Важное следствие номер два*

Если показать непрограммисту экран с интерфейсом пользователя, который на вид готов и красиво выглядит, то он решит, что программа почти готова.

Непрограммисты просто смотрят на экран и видят там пикселы. И если пикселы выглядят как программа, которая что-то делает, то они думают: да много ли еще нужно, чтобы она *действительно работала*?

Риск здесь в том, что если сначала смастерить интерфейс, например, чтобы легче было общаться с клиентом, то все решат, что работа уже почти закончена. А если вы еще год после этого будете заниматься «подводной частью», то никто не увидит того, что вы делаете, и все будут думать, что вы не делаете ничего.

## *Важное следствие номер три*

Интернет-компания, у которой есть феерический веб-сайт, состоящий из четырех или около того страниц, ценится дороже, чем компания с очень функциональным веб-сайтом, предоставляющим архивы за последние 3700 лет, но со стандартным серым фоном.

Ах, вспомнил: доткомы больше не стоят ничего. Тогда пошли дальше.

## *Важное следствие номер четыре*

Если проект будут «утверждать» заказчики или не испорченные техническим образованием менеджеры, дайте им несколько образцов графического дизайна, и пусть они выбирают.

Переставьте местами какие-нибудь элементы, поменяйте стиль и шрифты, подвигайте логотип компании, сделайте его побольше или поменьше. Пусть они почувствуют свою важность, и дайте им для этого какой-нибудь мишуры, с которой они будут возиться. На вашем графике работы это едва ли отразится. Хороший оформитель интерьеров всегда принесет своим клиентам образцы того, сего, третьего, чтобы им было из чего выбрать. Но он никогда не станет обсуждать с клиентом, где поставить посудомоечную машину. Она будет стоять рядом с раковиной, какими бы ни были пожелания клиента. Нет смысла тратить время, обсуждая, куда поставить посудомоечную машину, она должна стоять рядом с раковиной, и не нужно даже *упоминать об этом*; пусть клиент удовлетворяет свою потребность что-то проектировать каким-нибудь невинным способом, например, меняя 200 раз свое решение о том, чем покрывать прилавки – итальянским гранитом, мексиканской плиткой или норвежским деревом.

## *Важное следствие номер пять*

Если вам нужно произвести впечатление ожидаемыми результатами, то единственное, что имеет значение для успеха, это снимки экранов. Они должны быть как можно красивее.

И не надейтесь, что *кого-то увлекут ваши рассказы о том, как замечательно все будет работать*. Не думайте, что обратят внимание на функциональность. Нет. Они хотят видеть красивые пикселы.

Стив Джобс это понимает. Он чертовски хорошо понимает это. В Apple научились делать вещи, из которых получаются великолепные скриншоты,

например роскошные новые значки с разрешением 1024×1024 в панели, пожирающие драгоценное рабочее пространство. А пользователи десктопов Linux с ума сходят от полупрозрачных окошек терминала, которые красиво выглядят на скриншотах, но работать с ними неприятно. Каждый раз, когда объявляют о новом релизе Gnome или KDE, я сразу смотрю их скриншоты и говорю: «Ну, просто новая планета, Сатурн вместо Юпитера. Круто». Не все ли равно, что они сделали в действительности.

## *Управляйте айсбергом*

Помните того большого начальника в начале главы? Он был недоволен, потому что его программисты в начале показали ему прекрасные презентации PowerPoints – макеты, созданные даже не в VB, а в *Photoshop*. А теперь, когда они занялись подводной частью, ему кажется, что они ничего не делают.

Как с этим справиться? Легко, если вы поняли секрет айсберга. Помните, что все презентации, которые вы проводите с проектором в полутемной комнате, касаются *только пикселов*. Если сумеете, нарисуйте свой UI так, чтобы незавершенные части и выглядели незавершенными. Например, поставьте в панели инструментов какие-нибудь каракули вместо значков, если соответствующая функциональность еще не готова. Если вы строите веб-сервер, попробуйте не помещать на первой странице ссылки на функции, которые еще не реализованы. Тогда люди увидят, что сначала на вашей домашней странице было 3 команды, а потом их количество выросло до 20 по мере того, как вы реализовали другие возможности.

Еще важнее следить, чтобы люди представляли, как выполняется график. Создайте подробный график в формате Excel.[1] Еженедельно посылайте себе автоматизированное почтовое сообщение с сообщением, что вы выполнили уже не 32, а 35% плана и *успеете* выпустить продукт к 25 декабря. Определять, с должной ли скоростью подвигается вперед проект, нужно по фактам, а не по чьим-то ощущениям. И не разрешайте своему боссу играть в гольф титановыми клюшками Callaway, как бы вы ни хотели, чтобы он выиграл, потому что это нечестно, и американская ассоциация гольфа запретила их.

[1] См. главу 9.

ГЛАВА ДВАДЦАТЬ ШЕСТАЯ

# Закон дырявых абстракций

11 НОЯБРЯ 2002 ГОДА, ПОНЕДЕЛЬНИК

В технологии Интернета есть один удивительный и важный элемент, с которым вы сталкиваетесь ежедневно. Это протокол TCP, один из главных конструктивных элементов Интернета.

TCP дает возможность *надежно* передавать данные. Это означает, что если посылать сообщение по сети с помощью TCP, то оно дойдет до адресата и не будет при этом подделано или искажено.

С помощью TCP мы делаем многое – и грузим веб-страницы, и отправляем почту. Именно благодаря надежности TCP все увлекательные письма от мошенников из восточной Африки доходят до нас в полной сохранности. Какая радость.

Для сравнения есть другой метод передачи данных, IP[1], который *ненадежен*. Он не гарантирует доставку данных, и они могут перемешаться по пути. Послав группу сообщений через IP, не удивляйтесь, если до места дойдет только половина и часть из них окажется не в том порядке, в каком они были отправлены, а другие будут заменены какими-то посторонними, возможно, с фотографиями очаровательных детенышей орангутанга, но, скорее всего, с нечитаемым мусором, похожим на строку темы сообщения в спаме из Тайваня.

---

[1] Строго говоря, здесь неточность: сам протокол IP просто не предназначен для этого и не позволит вам пересылать какие-либо сообщения вообще. Здесь уместнее говорить, наверное, о протоколе UDP, который, так же как и TCP, работает над IP и к которому применимы все рассуждения автора. – *Примеч. науч. ред.*

А удивительно вот что: TCP построен поверх IP. Иными словами, TCP вынужден обеспечить надежную передачу данных *с помощью ненадежного средства*.

Чтобы показать, что это действительно волшебство, рассмотрим следующий морально эквивалентный, хотя и абсурдный сценарий из жизни.

Представим себе, что мы переправляем актеров с Бродвея в Голливуд, сажая их в машины и везя через всю страну. Некоторые машины разбиваются, и несчастные актеры гибнут. Иногда актеры по дороге напиваются и бреются наголо или делают татуировку на носу, из-за чего становятся слишком уродливы, чтобы работать в Голливуде, а иногда актеры приезжают не в том порядке, в котором выезжали, потому что каждый выбирает свою дорогу. Теперь представим себе, что появилась новая служба, «Голливудский экспресс», доставляющая актеров в Голливуд и гарантирующая, что они (a) доедут (b) по порядку и (c) в отличном состоянии. Чудеса заключаются в том, что у «Голливудского экспресса» нет иного способа доставки актеров, кроме ненадежного – сажать их в машины и вести через всю страну. Метод «Голливудского экспресса» заключается в том, что он проверяет состояние каждого прибывшего актера, и если оно не идеально, то звонит в головной офис и требует, чтобы послали однояйцового близнеца актера. Если актеры прибывают в неверном порядке, «Голливудский экспресс» переставляет их правильным образом. Если крупный НЛО по пути в район 51 потерпит аварию на дороге в Неваде, выведя ее из строя, все актеры, поехавшие этой дорогой, перенаправляются через Аризону, и «Голливудский экспресс» даже ничего не сообщает о случившемся режиссерам в Калифорнии. Последним лишь кажется, что актеры стали добираться чуточку дольше, чем обычно, а о крушении НЛО они и вовсе ничего *не слышали*.

Вот в чем, примерно, состоит чудесная механика TCP. Специалисты по ИТ часто называют это *абстракцией* – упрощением чего-то значительно более сложного, незримо происходящего в глубине. Оказывается, программирование компьютеров в значительной части состоит из построения абстракций. Что такое библиотека строковых функций? Это способ симулировать умение компьютеров манипулировать строками так же легко, как числами. Что такое файловая система? Это способ представить жесткий диск не рядом вращающихся магнитных дисков, способных хранить биты в определенных адресах, а некоей иерархической системой вложенных одна в другую папок, содержащих отдельные файлы, в свою очередь состоящие из одной или нескольких строк байтов.

Вернемся к TCP. Выше я ради простоты сказал небольшую неправду и чувствую, как кое у кого пар идет из ушей, потому что эта неправда выводит их из себя. Я сказал, что TCP гарантирует доставку сообщения. На самом деле это не так. Если ваша ручная змейка прогрызла кабель, соединявший ваш компьютер с Интернетом, и пакеты IP не могут больше по нему проходить, то TCP не поможет, и сообщение не пройдет.[1] Если вы нагрубили системным администраторам своей компании, и они наказали вас, подключив к перегруженному сетевому концентратору, то лишь немногие из ваших пакетов IP прорвутся через него, и TCP будет работать, но связь станет очень медленной.

Это то, что я называю *дырявой абстракцией*. TCP пытается обеспечить полную абстракцию от лежащей под ним ненадежной сети, но иногда сеть просачивается сквозь эту абстракцию, и вы ощущаете на себе то, от чего абстракция не смогла полностью защитить вас. Это лишь один пример того, что я окрестил «Законом дырявых абстракций»:

**Все нетривиальные абстракции в какой-то мере дырявы.**

Абстракции могут отказывать. Иногда чуть-чуть, иногда значительно. В них появляются течи. Все идет наперекосяк. Это происходит всегда, когда имеешь дело с абстракциями. Вот несколько примеров.

- Скорости обхода большого двумерного массива по горизонтали и по вертикали могут сильно отличаться, т. к в одном направлении могут намного чаще встречаться отсутствующие в памяти страницы, а загрузка страниц происходит медленно. Даже программирующим на ассемблере разрешается считать, что память – это одно большое плоское адресное пространство, но виртуальная память – это всего лишь абстракция, дающая течь, если страницы нет в физической памяти, и тогда одни выборки из памяти занимают больше времени, чем другие.[2]

---

[1] Более того, именно применительно к TCP – в этом случае можно очень долго (или бесконечно долго) ожидать отправки сообщения и не получить сообщения об ошибке. Когда приходится говорить об этом тем, кто поверхностно знаком с TCP, но наслышан о том, что это «протокол надежного соединения», то поднимается буря негодования. – *Примеч. науч. ред.*

[2] Это справедливо только для ОС с виртуализацией (отображением) страниц памяти на диск (свопингом). Но не для MS-DOS (ввиду ее «неразвитости») и не для ОС реального времени (т. к. свопинг там недопустим). – *Примеч. науч. ред.*

- Язык SQL должен позволять абстрагироваться от тех процедур, которые надо осуществить при выполнении запроса к базе данных, чтобы дать вам возможность просто указать, что вам нужно, а решать, какие процедурные действия требуются для выполнения запроса, будет сама база данных. Но иногда определенные запросы SQL выполняются в тысячи раз медленнее, чем другие, логически эквивалентные им. Известный пример: некоторые серверы SQL значительно быстрее выполняют запрос с условием «where a=b and b=c and a=c», чем запрос «where a=b and b=c», хотя результирующая выборка одинакова. Предполагается, что вас должна заботить спецификация, а не то, какими методами она осуществляется. Но иногда абстракция дает течь и приводит к ужасающему падению производительности, и тогда вам приходится вмешиваться в работу анализатора плана запроса и смотреть, что он сделал нехорошо, и придумывать, как заставить запрос выполняться быстрее.
- Несмотря на то, что сетевые библиотеки типа NFS и SMB позволяют работать с файлами на удаленных машинах, «как если бы» они были локальными, иногда связь становится очень медленной или вообще прекращается, и файл перестает вести себя так, будто он локальный, и программист вынужден писать код, который бы это учитывал. Абстракция «удаленный файл – это все равно, что локальный файл» оказывается дырявой.[1] Вот конкретный пример для системных администраторов UNIX. Если вы поместите личные каталоги пользователей на диск, смонтированный через NFS (одна абстракция), а ваши пользователи создадут файлы *forward*, чтобы их почта пересылалась в какое-то другое место (еще одна абстракция), то, когда NFS-сервер окажется недоступным во время получения новой почты, сообщения не будут переадресованы, потому что невозможно будет найти файл *forward*. Дырявость абстракции привела к потере нескольких почтовых сообщений.
- Классы строк C++ должны позволить вам считать, что строки ничем не хуже остальных типов данных. Они пытаются абстрагировать то обстоятельство, что работать со строками трудно,[2] и дать вам возможность обращаться с ними с такой же легкостью, как с целыми числами. Почти все строковые классы C++ перегружают оператор +,

[1] См. главу 17.

[2] См. главу 2.

благодаря чему конкатенацию можно выполнять с помощью s + "bar". Но знаете, что интересно? Как бы вы ни старались, никакой класс строк C++ не позволит вам написать "foo" + "bar", потому что строковые литералы всегда имеют в C++ тип указателей char*, а не строковый.[1] Абстракция дала течь, которую язык не позволяет заткнуть. (Забавно, что эволюцию C++ можно описать как историю попыток заткнуть дыры в абстракции строк. Почему нельзя было просто добавить собственный класс строк в сам язык, я вспомнить не могу.)

- Когда идет дождь, нельзя ехать так же быстро, как в сухую погоду, несмотря на то, что у вашей машины есть дворники на ветровом стекле, и фары, и крыша, и обогреватель, и все это позволяет вам не беспокоиться о том, что идет дождь (абстрагирует вас от погоды), но, однако же, вы должны помнить об опасности гидропланирования (а в Англии – аквапланирования), а иногда дождь усиливается настолько, что видимость уменьшается, и в дождь приходится ехать медленнее, потому что полностью абстрагироваться от погоды нельзя в силу «Закона дырявости абстракций».

Закон дырявых абстракций сомнителен в частности потому, что из него следует, что абстракции на самом деле не настолько облегчают нам жизнь, как это предполагалось. Когда я обучаю программистов языку C++, мне не хочется касаться char* и арифметики указателей. Замечательно было бы сразу перейти к строкам STL. Но в один прекрасный день они пишут код "foo" + "bar", и начинаются весьма странные вещи, и тогда мне приходится останавливаться и все-таки объяснять им все про char*. А как-нибудь в другой раз они пытаются вызвать функцию Windows API, у которой, согласно описанию, есть аргумент OUT LPTSTR, и они не смогут обращаться к ней, пока не разберутся с char*, указателями, Unicode, wchar_t, файлами заголовков TCHAR и всем прочим, что вытекает сквозь прорехи абстракций.

При обучении программированию COM было бы чудесно показать, как пользоваться мастерами в Visual Studio и всякими возможностями для генерации кода, но когда возникают неполадки, человек не имеет ни малейшего представления о том, что могло случиться, как найти и исправить ошибку. И мне приходится рассказывать все о IUnknown, и CLSID, и ProgIDS, и... о, небо!

---

[1] Имеется в виду, что даже если определить еще одну функцию класса – оператор неявного преобразования типа из char* в string, то в указанном выражении компилятор задействует операцию адресной арифметики как более простую. – *Примеч. науч. ред.*

Как было бы прекрасно, если бы при обучении ASP.NET достаточно было показать, что можно дважды щелкнуть по объекту, а затем написать код, который будет выполняться на сервере, когда по этим объектам щелкнет пользователь. Действительно, ASP.NET абстрагирует различие между написанием кода HTML для обработки щелчка по гиперссылке (<a>) и кода для обработки щелчка по кнопке. Загвоздка состояла в том, что проектировщикам ASP.NET требовалось скрыть, что в HTML нельзя передать форму через гиперссылку. Они сделали это, сгенерировав несколько строчек JavaScript и прикрепив к гиперссылке обработчик onclick. Однако эта абстракция с прорехой. Если у конечного пользователя отключен JavaScript, то приложение ASP.NET работает некорректно, а если программист не знает, от чего абстрагируется ASP.NET, то просто не сможет понять, что произошло.

Следствие из закона дырявых абстракций: когда некто предлагает новое волшебное средство генерации кода, призванное неслыханно повысить эффективность нашего труда, многие говорят: сначала научись делать это вручную, а уж потом пользуйся этим волшебным инструментом для экономии времени. Средства генерации кода, претендующие на то, чтобы абстрагировать нас от чего-то, оказываются дырявыми, как и все абстракции, и единственный способ компетентно справляться с дырами, это изучить, как действуют эти абстракции и от чего они нас абстрагируют. Поэтому абстракции дают возможность тратить меньше времени на работу, но не дают возможности тратить меньше времени на учебу.

Все это приводит к парадоксу: средства программирования поднимаются на все более высокий уровень, и абстракции становятся все лучше, а стать искусным программистом становится все труднее.

Во время моей первой стажировки в Microsoft я писал библиотеки строковых функций для Mac OS. Типичное задание: написать вариант strcat, который возвращал бы указатель на конец новой строки. Несколько строчек кода C. Я делал все точно по K&R – одна тонкая книга по языку программирования C.

Сегодня для работы над CityDesk мне надо знать Visual Basic, COM, ATL, C++, InnoSetup, внутренности Internet Explorer, регулярные выражения, DOM, HTML, CSS и XML – все это инструменты более высокого уровня по сравнению со старым материалом K&R, но я все равно должен знать и то, что написано в K&R, иначе мне конец.

Десять лет назад мы мечтали о том, что новые парадигмы программирования постепенно сделают его проще. И действительно, созданные за эти годы абстракции *позволяют* справляться со сложностью в разработке

программного обеспечения такого порядка, с каким не приходилось сталкиваться десять или пятнадцать лет назад, например с программированием GUI или сетевым программированием. И хотя эти замечательные инструменты, такие как современные языки, основанные на ОО-формах, позволяют невероятно быстро выполнять огромную работу, в один прекрасный день приходится разбираться, где абстракция дала течь, и на это уходит две недели. И когда вам нужен программист, чтобы писать преимущественно на VB, то желательно, чтобы он был больше чем просто VB-программистом, потому что иначе он будет беспомощно вязнуть всякий раз, когда абстракция VB дает течь.

Закон дырявых абстракций тянет нас вниз.

ГЛАВА ДВАДЦАТЬ СЕДЬМАЯ

# Лорд Пальмерстон о программировании

11 ДЕКАБРЯ 2002 ГОДА, СРЕДА

Когда-то достаточно было прочесть одну книгу Питера Нортона,[1] чтобы узнать буквально все о программировании IBM-PC. Последние 20 лет программисты всего света отчаянно трудились, возводя одну абстракцию над другой поверх IBM-PC, чтобы повысить его мощь и облегчить программирование.

Но согласно Закону дырявых абстракций[2] один лишь объем материала, который надо изучить, чтобы стать хорошим программистом, неуклонно увеличивается, хотя создание абстракций и призвано упростить программирование.

Чтобы стать специалистом, настоящим экспертом лишь в отдельной области программирования, нужны годы. Конечно, есть масса толковых подростков, которые за неделю осваивают Delphi, еще за неделю Python, еще за неделю Perl, после чего они считают себя знатоками. При этом у них нет даже самого отдаленного представления о том, сколь многого они не знают.

Я стал работать с ASP и VBScript с того самого времени, как только они появились. VBScript – самый миниатюрный язык на свете, а программирование ASP заключается в том, чтобы изучить штук пять классов, из которых часто применяются лишь два. И лишь сейчас у меня, наконец, появилось ощущение, что я научился строить приложения ASP/VBScript оптималь-

---

1 Norton, Peter (Питер Нортон) «Inside the IBM PC: Access to Advanced Features and Programming», R. J. Brady Co., 1983.

2 См. главу 26.

ным образом. Мне кажется, что я наконец-то знаю, в каком месте лучше всего поместить код для доступа к базе данных, как лучше всего применять ADO для получения результирующих наборов записей, как лучше всего разделить HTML и код и т. д. И я, наконец, стал применять регулярные выражения вместо одноразовых функций для действий со строками. Лишь на прошлой неделе я научился выгружать COM-объекты из памяти, чтобы их можно было перекомпилировать (без перезагрузки всего веб-сервера).

Fog Creek слишком маленькая компания, чтобы иметь узких специалистов. Поэтому, когда мне потребовалось написать хороший инсталлятор[1] для FogBUGZ,[2] нашего продукта, основанного на ASP/VBScript, я положился на свой многолетний опыт работы с C++/MFC, годы работы с различными Windows API, хорошие навыки в работе с Corel PHOTO-PAINT и сделал аккуратную картинку в углу окна мастера. Затем, чтобы FogBUGZ могла идеально работать с Unicode, я написал небольшой элемент ActiveX с помощью C++ и ATL, для чего привлек свой многолетний опыт работы с C++ и COM, а также знание о кодировках, которые я в течение недели изучал, когда писал код для CityDesk.

В другой раз, когда обнаружилась странная ошибка, встречавшаяся только в NT 4.0, я отладил ее за три минуты, потому что я умел работать с VMWare, у меня была чистая система NT 4.0, установленная в VMWare, я умел выполнять удаленную отладку в Visual C++ и я знал, что надо прочитать регистр EAX, чтобы получить значение, возвращаемое функцией. Тот, для кого все это в новинку, потратил бы, наверное, не меньше часа на решение такой проблемы, но я знал огромный объем «материала», который я фактически изучаю с 1983 года, когда у меня появился первый IBM-PC и та самая книга Нортона.

Дырявые абстракции означают, что кривая обучения у нас напоминает по форме клюшку: изучить 90% необходимого в повседневной работе материала можно за неделю. Но на освоение оставшихся 10% может уйти пара лет. Вот тут и проявляется во всем блеске превосходство действительно опытных программистов над теми, кто говорит, что если он чего-то не знает, то возьмет книжку и прочтет. Если вы собираете команду, то нет ничего страшного, если в ней окажется много не слишком опытных программистов, которые станут мастерить большие блоки кода с помощью абстрактных инструментов, но команда окажется неработоспособной, если в ней

[1] См. *www.joelonsoftware.com/news/20021002.html*.

[2] См. *www.fogcreek.com/FogBUGZ*.

не будет нескольких действительно опытных членов, которые смогут делать действительно трудные вещи.

Есть множество областей программирования, и в каждой из них настоящим экспертом можно стать, только овладев уймой знаний. Вот три области, которые лично я знаю лучше всего:

- MFC/C++/Windows
- VBScript/ASP
- Visual Basic

Все то, что по существу и называется Windows-программированием. Да, я писал код для UNIX и код на Java, но не так много. Мой профессионализм в Windows-программировании состоит из знания не только базовых технологий, но и всей вспомогательной инфраструктуры. И я утверждаю, что хорошо владею Windows-программированием, потому что я также знаю COM, ATL, C++, ассемблер 80x86, Windows API, IDispatch (OLE Automation), HTML, DOM, объектную модель Internet Explorer, внутреннее устройство Windows NT и Windows 95, LAN Manager и сетевое взаимодействие NT (в том числе вопросы безопасности – ACE, ACL и все такое прочее), SQL и SQL Server, Jet и Access, JavaScript, XML и некоторые другие вопросы, составляющие альфу и омегу программирования и вселяющие оптимизм в того, кто постиг их суть. Когда мне не удается заставить функцию `StrConv` в VB сделать то, что мне нужно, я могу накропать элемент COM, чтобы воспользоваться C++ с ATL и вызывать функции `Mlang`, нисколько не сбиваясь при этом с ритма. Мне потребовались годы, чтобы достичь этого.

Есть множество других областей программирования. Например, область, в которой пишут для BEA WebLogic и где нужно знать J2EE, Oracle и тьму относящихся к Java вещей, которые я даже не смогу перечислить. Есть сильные разработчики для Макинтошей, владеющие программированием CodeWarrior, MPW, Toolbox в System 6 – System X, Cocoa, Carbon и даже таким восхитительным антиквариатом, как OpenDoc, теперь уже бесполезным.

Однако очень немногие разбираются более чем в одной или двух областях, потому что знать надо так много, что, не проработав в одной из этих областей не менее пары лет, их не освоить по-настоящему.

Но учеба – это необходимость.

Люди часто злятся, если приходят на собеседование по поводу приема на работу и им отказывают из-за того, скажем, что у них нет опыта работы с Win32 (или J2EE, или программирования в Mac OS, или еще чего-нибудь). Или их раздражает, когда какой-нибудь идиот-агент по найму, который

не узнает MSMQ, если она укусит его за копчик, звонит им и спрашивает, есть ли у них «десять лет опыта работы с MSMQ».

Тот, кто хоть какое-то время не занимался программированием под Windows, может думать, что Win32 – это просто некая библиотека, и что ее, как всякую другую библиотеку, можно изучить, почитав книжку, а потом вызывать функции Win32, когда это потребуется. Кто-то может думать, что основы программирования, скажем, уровень его знаний в C++, составляют 90%, а всякие там API – это жалкие 10%, которые осваиваются за несколько недель. Так вот: времена изменились. Сегодня соотношение обратное.

Сегодня очень немногим приходится работать над низкоуровневыми алгоритмами C, которые жонглируют байтами. Мы теперь почти все вызываем API, а не жонглируем байтами.[1] Тот, кто фантастически хорошо владеет C++ и не имеет опыта работы с API, знает лишь 10% того, что необходимо почти ежедневно для написания кода, выполняемого поверх API. Если в экономике все в порядке, то это не имеет значения.[2] Вы можете найти работу, и ваш наниматель оплатит то время, которое необходимо, чтобы вы в достаточной мере освоили конкретную платформу. Но когда в экономике спад и на каждую открывшуюся вакансию претендует 600 человек, работодатели могут позволить себе выбирать таких программистов, которые уже достигли уровня экспертов в конкретной платформе. Например, программистов, способных назвать четыре способа передачи файла по FTP из кода Visual Basic и указать на достоинства и недостатки каждого из них.

Площадь поверхности всех этих областей программирования огромна, и на ней разгораются бессмысленные жаркие войны, участники которых доказывают, что их область лучше. Некто поместил на моем дискуссионном форуме следующий самодовольный комментарий:

> А вот еще одна причина, по которой я счастлив, что живу в «свободном мире». Это свобода слова (ну, почти) и свобода от необходимости участвовать в создании программ установки и реестра, не говоря о многом другом.

По-видимому, человек хотел сказать, что в мире Linux не пишут программ установки. Не хочется вас разочаровывать, но у вас есть вещи, не ме-

[1] См. *www.joelonsoftware.com/articles/fog0000000250.html*.

[2] См. *www.joelonsoftware.com/articles/fog0000000050.html*.

нее сложные: все эти imake, make, файлы *config* и т. д., а закончив код, вы еще распространяете приложения с 20-килобайтным файлом *INSTALL*, полным мудрых указаний типа «Вам понадобится zlib» (*это что такое?*) или «Это может занять довольно много времени. Сходите за рантами» (ранты, надо думать, – это некая разновидность леденцов). А реестр? Вместо одного большого организованного улья с парами имя/значение тысячи разных форматов файлов, свой у каждого приложения, и все эти *whateverrc* и *foo.conf* разбросаны по всем местам. А чтобы изменить настройки Emacs, надо научиться программировать на lisp, и каждая командная оболочка требует, чтобы вы изучили ее собственный диалект языка сценариев оболочки, если потребуется изменить настройки, и т. д. и т. п.

Те, кто знает только одну область, перестают критически относиться к ней, и когда они слышат об осложнениях, существующих в другой области, то думают, что в их-то собственной области осложнений нет. Они есть. Просто вы преодолели их благодаря тому, что глубоко в них разобрались. Эти области стали настолько велики и сложны, что лучше прекратить их сравнивать. Лорд Пальмерстон сказал: «Проблема Шлезвиг-Гольштейна настолько сложна, что всего три человека в Европе разобрались в ней. Одним из них был принц Альберт, ныне покойный. Вторым был немецкий профессор, сошедший с ума. Третьим был я, но я все об этом позабыл». Эти программные миры настолько велики, сложны и многогранны, что когда разумные в прочих отношениях люди помещают в блогах пустые высказывания типа «У Microsoft плохие операционные системы», они выглядят просто неумными. Можно ли так просто подвести итог миллионам строк кода с сотнями крупных отдельных областей, созданным тысячами программистов за пару десятилетий, где никому в отдельности не разобраться даже в сколько-нибудь крупной отдельной части? Я не защищаю Microsoft, я просто хочу отметить, что бессмысленная трата времени на крупные легкомысленные обобщения, совершаемые с позиций глубокого невежества, приобрела огромные масштабы в Сети.

Я размышлял о том, как выпускать приложения для Linux, Macintosh и Windows, чтобы версии для Linux и Macintosh не оказывались неоправданно дорогими. Для этого нужна некая кроссплатформенная библиотека.

Такая попытка была сделана в Java, но Sun не удалось создать такие GUI, чтобы получаемые приложения выглядели достаточно натурально на каждой платформе. Примерно так же в «Звездном походе» пришелец, разглядывавший землю в телескоп, точно знал, как должна *выглядеть* человече-

ская пища, но не знал, что у нее еще должен быть какой-то *вкус*.[1] У приложений Java меню находятся в правильном месте, но на клавиши они реагируют не так, как все другие приложения Windows, а диалоговые окна с закладками выглядят жутковато. И как бы вы ни старались, панели меню не будут выглядеть в точности, как панели меню Excel. Почему? Потому что Java не одобряет использование собственных средств платформы, если абстракции оказывается недостаточно. Если вы программируете в AWT, то не сможете узнать HWND окна, не сможете обращаться к Microsoft API и уж точно не сможете перехватить WM_PAINT и сделать все по-своему. И Sun достаточно хорошо разъяснила, что если вы попытаетесь это сделать, то лишитесь Чистоты.[2] Вы Испорчены и пропадайте пропадом.

После ряда широко освещенных в прессе провалов построения GUI на Java (например, набор Corel Java Office и Netscape Javagator) многие стараются держаться подальше от этой области. В Eclipse[3] построили заново собственную оконную библиотеку, взяв родные графические элементы платформы, благодаря чему можно писать приложения Java, выглядящие вполне естественно.

Конструкторы Mozilla подошли к проблеме кросс-платформенности с помощью собственного изобретения под названием XUL. На меня произвело впечатление то, что им уже удалось. Mozilla все-таки добралась до той точки, где есть не только вид, но и вкус настоящей еды. Даже моя любимая бука, Alt+Space N, сворачивающая окно активной программы, действует в Mozilla. Времени они потратили немало, но с результатом.

Митч Капор (Mitch Kapor), основатель Lotus и создатель 123, решил, что в его следующем приложении поддержку кросс-платформенности будут обеспечивать системы под названием wxWindows[4] и wxPython.[5]

---

1 «The Squire of Gothos» – телесериал «Star Trek» (Paramount Studio, January 12, 1967). Можно прочесть полный текст, как поэму, на *www.voyager.cz/tos/epizody/19squireofgothostrans.htm*.

2 Вообще-то Sun по этому вопросу разъяснила только то, что если бы в Java были допущены такие «платформенные» вольности, то это положило бы конец какой-либо защищенности приложений Java, которые как никакие другие ориентированы на работу в сетях. – *Примеч. науч. ред.*

3 См. *www.eclipse.org/*.

4 К настоящему времени wxWindows из-за лицензионных разногласий переименована в wxWidgets. – *Примеч. науч. ред.*

5 См. *blogs.osafoundation.org/mitch/000007.html*.

Так что же лучше: XUL, Eclipse SWT или wxWindows? Не знаю. Все это такие обширные области, что я не смог оценить их по-настоящему и высказать свое мнение. Одного чтения руководства недостаточно. Надо попотеть в каждой год-другой, и лишь тогда вы выясните, что вещь стоящая или что при всем желании вы не сможете придать своему UI вкус настоящей пищи. К сожалению, в большинстве проектов приходится решать, к какому миру он будет относиться, до того как будет написана первая строка кода, то есть тогда, когда у вас меньше всего информации. У нас была работа, в которой пришлось столкнуться с довольно скверной архитектурой, потому что первые программисты в этом проекте использовали его одновременно для изучения программирования в C++ и Windows. Древнейший код был написан без всякого представления о событийно-управляемом программировании. Базовый класс строк (у нас, разумеется, был собственный класс строк) мог быть включен в учебник в качестве примера всех ошибок, которые можно сделать при проектировании класса C++. В конечном итоге мы подчистили и переработали значительную часть старого кода, но некоторое время он нам досаждал.

Итак, на сегодняшний день мой совет следующий: не беритесь за новый проект, не располагая хотя бы одним архитектором с несколькими годами серьезной работы с языком, классами, API и платформами, для которых вы делаете продукт. Если платформу можно выбирать, берите ту, с которой лучше всего знакомы ваши программисты, даже если она не самая модная или номинально наиболее продуктивная. А если вы разрабатываете абстракции или инструменты программирования, не пожалейте труда, чтобы залатать прорехи в них.

ГЛАВА ДВАДЦАТЬ ВОСЬМАЯ

# Оценки производительности труда

15 ИЮЛЯ 2002 ГОДА, ПОНЕДЕЛЬНИК

«Спасибо, что обратились в Amazon.com, чем могу помочь?» И – щелк! Вас отключили. Это раздражает. Вы только что прождали 10 минут, чтобы пообщаться с человеком, и вас почему-то вдруг сразу рассоединили.

Почему? Как сообщил Майк Дэйзи, в Amazon оценивали сотрудников, отвечающих на звонки клиентов, по количеству вызовов, принятых ими в час.[1] Легче всего было поднять оценку своего труда, кладя трубку и тем самым увеличивая количество вызовов, принимаемых в час.

Скажете, это нетипично?

Когда Джефф Вейцен возглавил Gateway, он ввел новую политику для экономии на звонках в службу поддержки. «Те, кто разговаривал с клиентом дольше 13 минут, не получали месячной премии, – пишет Катрина Брукер.[2] – В результате сотрудники стали всеми силами избавляться от клиентов: делать вид, что линия неисправна, вешать трубку или, зачастую с большими затратами, посылали им новые запчасти или компьютеры. Неудивительно, что показатели удовлетворенности клиентов Gateway, некогда лучшие в отрасли, упали ниже среднего».

Складывается впечатление, что каждый раз, когда вы пытаетесь измерить производительность труда работников индустрии знаний, все сразу рушится, и вы получаете то, что Роберт Д. Остин (Robert D. Austin) назвал *дисфункцией измерения* (*measurement dysfunction*). Его книга «Measuring

---

1 Daisey, Mike (Майк Дейзи) «21 Dog Years: Doing Time@Amazon.com», Free Press, 2002.

2 Business 2.0, April 2000.

and Managing Performance in Organizations»[1] представляет собой превосходное и глубокое исследование предмета. Администраторы любят реализовывать системы измерений и любят привязывать вознаграждение к производительности, измеряемой на основании этих систем. Но если только они не находятся под неусыпным надзором, работники получают стимул трудиться ради оценки и интересуются исключительно этой оценкой, а не реальной ценностью или качеством своего труда.

Часто поощряют тех программистов, которые a) пишут много кода и b) исправляют много ошибок. В этом случае проще всего добиться успеха, выдавая как можно больше кода с ошибками и потом все исправляя и не тратя время на отлаженный код. Если вы попытаетесь с этим бороться и станете наказывать программистов за ошибки, то они станут скрывать их или не сообщат тестерам, что написали новый код, в надежде, что будет найдено меньше ошибок. Вы все равно проиграете.

Руководителей компаний Fortune 500 обычно поощряют основным окладом, к которому добавляется опцион на акции. Опционы на акции часто стоят десятков или сотен миллионов долларов, по сравнению с которыми базовый оклад оказывается несущественным. В результате эти руководители делают все от них зависящее, чтобы взвинтить цену акций на бирже, иногда ценой банкротства и гибели компании (как мы видим снова и снова в новостных заголовках месяца). Они будут делать это, даже если акции поднимутся лишь на короткое время, и продадут их в момент максимума цены. Комиссии по возмещению убытков реагируют медленно, но их последней блестящей идеей было требование, чтобы руководящий работник держал акции, пока не уйдет из компании. Как страшно. Теперь возникает стимул временно поднять цену акций и уйти. И снова вы проиграете.

Не верьте мне на слово: возьмите книгу Остина, и вы поймете, почему ошибки в измерениях неизбежны, если вы не можете держать работников под полным контролем (а это почти всегда так).

Я давно утверждаю, что денежное стимулирование неэффективно,[2] даже если вы *можете* измерить, кто хорошо работает, а кто плохо, но Остин усиливает это положение, показывая, что даже измерить производительность нельзя, поэтому еще менее вероятно, что материальное стимулирование окажется эффективным.

---

[1] Austin, Robert (Роберт Остин) «Measuring and Managing Performance in Organizations» (Измерение производительности труда и управление ею), Dorset House,1996.

[2] См. главу 21.

ЧАСТЬ ТРЕТЬЯ

# Мысли Джоэла: случайные высказывания по не столь случайным поводам

ГЛАВА ДВАДЦАТЬ ДЕВЯТАЯ

# Рик Чэпмен в поисках глупости[1]

1 августа 2003 года, пятница

Во всех известных мне хайтек-компаниях идет война между гиками и пиджаками.

Прежде чем вы станете читать новую пропагандистскую книгу волшебника маркетинга программного обеспечения и сверхпиджака Рика Чэпмена, послушайте меня минутку и узнайте, что думают гики.

Минутка-то у вас найдется?

Представьте себе самого типичного бледного, пьющего джолт-колу, питающегося китайской кухней, играющего в видеоигры, читающего Slashdot, не вылезающего из командной строки Linux чудака. Поскольку это всего лишь стереотип, можете представлять себе его как угодно, коротышкой или пухленьким мальчиком, но в любом случае он *не из тех*, кто играет в футбол с ребятами из своей школы, когда приезжает к маме на День Благодарения. Кроме того, поскольку это стереотип, мне не надо долго извиняться за то, что я сделал его *таким*.

Вот как рассуждает наш стереотипный программер: «Microsoft делает более слабые продукты, но проводит более сильный маркетинг, поэтому все покупают у них».

Спросите его, как он относится к маркетинговым специалистам в его собственной компании. «Они просто тупы. Вчера в комнате отдыха я круп-

---

[1] Эта глава первоначально вышла в виде предисловия к книге Меррила Рика Чэпмена (Merrill Rick Chapman) «In Search of Stupidity: Over 20 Years of High-Tech Marketing Disasters» (В поисках глупости: 20 с лишним лет маркетинговых провалов высоких технологий), Apress, 2003.

но поспорил с этой глупой девицей из отдела продаж, и через десять минут выяснилось, что она *понятия не имеет*, какая разница между 802.11*a* и 802.11*b*. Фе!»

А скажи мне, милый друг, чем занимаются люди из маркетинга? «Без понятия. Наверное, играют с клиентами в гольф, когда не заставляют меня исправлять их идиотские таблицы спецификаций. Если бы это зависело от меня, всех бы уволил».

Славный малый по имени Джеффри Тартер ежегодно публиковал список 100 крупнейших производителей программного обеспечения для персональных компьютеров, который он называл «Soft-letter 100». Вот как выглядел этот список в 1984 году:[1]

| Место | Компания | Годовой доход |
|---|---|---|
| #1 | Micropro International | $60 000 000 |
| #2 | Microsoft Corp. | $55 000 000 |
| #3 | Lotus | $53 000 000 |
| #4 | Digital Research | $45 000 000 |
| #5 | VisiCorp | $43 000 000 |
| #6 | Ashton-Tate | $35 000 000 |
| #7 | Peachtree | $21 700 000 |
| #8 | MicroFocus | $15 000 000 |
| #9 | Software Publishing | $14 000 000 |
| #10 | Broderbund | $13 000 000 |

Отлично, Microsoft на втором месте, но она входит в группу компаний с примерно одинаковыми годовыми доходами.

А вот тот же список для 2001 года:

| Место | Компания | Годовой доход |
|---|---|---|
| #1 | Microsoft Corp. | $23 845 000 000 |
| #2 | Adobe | $1 266 378 000 |
| #3 | Novell | $1 103 592 000 |
| #4 | Intuit | $1 076 000 000 |
| #5 | Autodesk | $926 324 000 |

[1] Jeffrey Tarter, *Soft*letter*, April 30, 2001, 17:11.

| Место | Компания | Годовой доход |
|---|---|---|
| #6 | Symantec | $790 153 000 |
| #7 | Network Associates | $745 692 000 |
| #8 | Citrix | $479 446 000 |
| #9 | Macromedia | $295 997 000 |
| #10 | Great Plains | $250 231 000 |

О-о! Соблаговолите отметить: *все до единой компании*, исключая Microsoft, исчезли из первой десятки. Кроме того, обратите, пожалуйста, внимание, что Microsoft *настолько опережает* следующего по результатам игрока, что это даже не смешно. Adobe удвоила бы свою прибыль, если бы прибавила к ней то, что в Microsoft тратят на газированную воду.

Рынок программного обеспечения для персональных компьютеров – это и есть Microsoft. Доходы Microsoft, как можно посчитать, составляют 69% *всех 100 ведущих компаний, вместе взятых*.

Вот об этом мы здесь и говорим.

Это всего лишь более сильный маркетинг, как заявляет наш воображаемый гик? Или это результат противозаконной монополии? (Что вызывает вопрос: как Microsoft *удалось получить* такую монополию? Что-нибудь одно из двух.)

Рик Чэпмен утверждает, что ответ проще: Microsoft была единственной компанией в списке, ни разу не сделавшей глупой фатальной ошибки. Обязана ли она своим успехом интеллектуальному превосходству или лишь слепой удаче, но самой крупной ошибкой Microsoft была танцующая скрепка. И так ли уж плоха она была *на самом деле*? Мы смеялись над ней, отключали ее и все равно пользовались Word, Excel, Outlook и Internet Explorer ежеминутно и ежедневно. Но для всех других компаний, когда-то имевших лидерство на рынке и безвозвратно утративших его, можно указать одну или две крупных ошибки, обернувшихся потерей правильного курса и столкновением с айсбергом. Micropro начала возиться с переделкой архитектуры принтера, вместо того чтобы совершенствовать свой главный продукт WordStar. Lotus потратила полтора года, чтобы втиснуть 123 в машины с 640 килобайтами памяти; когда они справились с этим, появилась Excel, а от машин с памятью 640 Кбайт остались слабые воспоминания. Digital Research невероятно завысила цену на CP/M-86 и утратила

шанс стать стандартом де факто операционных систем для PC.[1] VisiCorp досудилась до собственного исчезновения. Ashton-Tate не упустила возможности прогнать разработчиков dBase, отравив непрочную экологию, столь необходимую для успеха поставщика платформы.

Конечно, я программист и склонен винить за эти глупые ошибки тех, кто занимается маркетингом. Почти все эти ошибки связаны с неспособностью людей бизнеса, не сведущих в технических вопросах, понять элементарные факты технологии. Когда преуспевший с Pepsi Джон Скалли стал продвигать Apple Newton, он не знал того, что известно каждому, кто получил образование в информатике: распознавание рукописного текста *невозможно*. Это происходило в то самое время, когда Билл Гейтс затаскивал программистов на совещания и упрашивал создать единый элемент управления rich text edit («расширенного редактирования текста»), который можно было бы повторно использовать во всех их продуктах. Посади Джима Манци (пиджака, который позволил бизнес-администраторам захватить Lotus) на такое совещание, он стал бы хлопать там глазами. «Что еще за элемент rich text edit?» Ему никогда не пришло бы в голову осуществлять техническое руководство, потому что он не «грокал» технологии; на самом деле само слово *grok* в этом предложении, вероятно, ошеломило бы его.

Если хотите знать *мое* мнение, а оно предвзятое, то *никакая софтверная компания не может добиться успеха*, если у руля в ней не стоит программист. Все живые до сих пор свидетельства подтверждают мой вывод. Но и сами программисты наделали массу тупых ошибок. Поразительное решение Netscape переписать заново свой броузер вместо того, чтобы совершенствовать старый код, обошлось им потерей нескольких лет в Интернете, в течение которых принадлежавшая им доля рынка сократилась с 90% до 4, и это была *идея программиста*. Конечно, ничего не понимавшее в технике неопытное руководство этой компании не понимало, *почему* это плохое решение. Тем не менее есть масса программистов, защищающих решение Netscape переписать все с самого начала. «Джоэл, старый код был действительно ужасен!» Да-да, ну-ну. Такими программистами нужно восхищаться за их любовь к хорошо сделанному коду, но их на 100 метров

[1] Как мы все знаем, причиной того, что «стандартом де факто операционных систем для PC стала MS-DOS», а не гораздо более совершенная CP/M-86, было не только (и не столько) то, что Digital Research невероятно завысила цену, но и коррумпированная и «личностная» возня внутри IBM периода принятия этого решения. Можно предполагать, что мы еще при своей жизни узнаем и более глубинные «геополитические» причины такого развития событий... – *Примеч. науч. ред.*

нельзя подпускать к принятию каких-либо деловых решений, потому что очевидно, что ясность кода для них важнее, чем, простите, выпуск продукта на рынок.

Поэтому я немного уступлю Рику и скажу, что если вы хотите добиться успеха в софтверном бизнесе, то ваши руководители должны хорошо знать и любить программирование, но они должны знать и любить бизнес тоже. Найти лидера с большими способностями в обоих этих направлениях трудно, но это единственный способ избежать одной из тех фатальных ошибок, которые Рик так любовно собрал в своей книге. Так что прочтите ее, посмейтесь, и, если вашей компанией руководит какая-нибудь тупая голова, приведите в порядок свое резюме и начните подыскивать жилище в Редмонде.

ГЛАВА ТРИДЦАТАЯ

# А какую работу делают собаки в вашей стране?

5 мая 2001 года

Насколько наивными мы *были*?

Мы предполагали, что Безос просто *реинвестировал* прибыль, поэтому ее не оказалось в окончательном итоге.

В прошлом году примерно в эти же дни первые сообщения о крупных провалах «доткомов» стали появляться в новостях. Boo.com. Toysmart.com. Тактика «быстрого превращения в крупную компанию» не работала. Пятьсот людей 31 года отроду в «докерсах» обнаружили, что простое копирование Джеффа Безоса – это еще не бизнес-план.

Последние две недели показались в Fog Creek необычно спокойными. Мы завершаем CityDesk. Мне бы хотелось рассказать вам о CityDesk, но этот рассказ пока придется отложить. Я должен рассказать вам о корме для собак.

Корм для собак?

В прошлом месяце Сара Корбетт рассказала нам о потерянных мальчиках,[1] беженцах из Судана в возрасте от 8 до 18 лет, разделенных со своими семьями и вынужденных совершить тысячемильный переход из Судана в Эфиопию и Кению. Половина из них умерла по дороге от голода, жажды, аллигаторов. Несколько человек были спасены и привезены в такие места, как, например, Фарго, Сев. Дакота, посреди зимы. «А в этих кустах есть

[1] Corbett, Sara (Сара Корбетт) «The Long Road From Sudan to America» (Долгий путь из Судана в Америку), *The New York Times Magazine*, April 1, 2001. Доступно на *wwwnytimes.com/2001/04/01/magazine/01SUDAN.html? pagewanted=all* (требуется бесплатная регистрация).

львы?» – спросил один из них, когда его везли на машине из аэропорта в его новый дом. И потом в супермаркете:

> Питер тронул меня за плечо. Он держал в руке банку корма для собак Purina – «Извини меня, Сара, но что это такое?» Полки за его спиной до самого потолка были уставлены кормом для домашних животных. «Э-э, это корм для наших собак», – ответила я, сжимаясь при мысли, как это воспримет человек, последние восемь лет питавшийся всякой дрянью. «А-а, понятно», – сказал Питер, с удовлетворенным видом вернув банку на полку. Он двинулся дальше, толкая перед собой тележку, но через несколько шагов обернулся ко мне с вопрошающим взглядом. «Скажи, пожалуйста, – спросил он, – а какую работу должны делать собаки в вашей стране?»

Собаки. Да, Питер. В Фарго хватает еды, даже для собак.

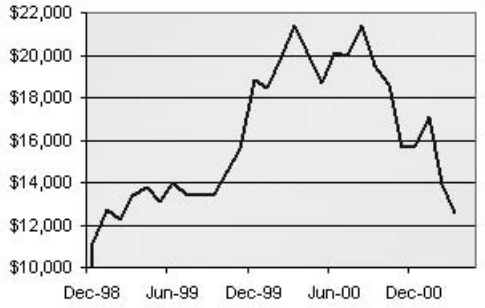
Это был унылый год.

А *началось* все так замечательно. Мы все кучей набились в B2B, B2C и P2P, как счастливое семейство залезает в свой Suburban, отправляясь в воскресенье покупать Krispy Kreme в Donut Shop. Но самое веселье было впереди; веселье началось, когда мы увидели, как проваливаются худшие бизнес-планы, и их акции падают с 316 до 3/16. Взгляните, балаболящие о новой экономике! Да, *злорадство*. Да, ликование, когда журнал «Wired» в очередной раз доказывает, что стоит ему поместить что-то на обложку, и всего через несколько месяцев обнаруживается, что это глупость и заблуждение.

А эту так называемую новую экономику «Wired» *действительно* взорвал, потому что к тому времени они должны были уже *знать*, каким смертельным поцелуем оказывается их обложка для любых технологий, компаний, коллективных идей, после того как они годами превозносили smell-orama, обреченные компании-производители игр и провозглашали приход PointCast на смену Web,[1]... нет, погодите, PointCast *уже* заменил Web в марте 1997 года. Но как бы то ни было, они искушали судьбу и не просто поместили Новую Экономику на обложку, они посвятили Новой Экономике *весь*

[1] Craig Bicknell (Крейг Бикнел) «PointCast Coffin About to Shut» (Крышка гроба PoinCast скоро захлопнется), *Wired News*, March 29, 2000. См. *www.wired.lycos.com/news/business/0,1367,35208,00.html*.

*чертов номер*,[1] в результате чего NASDAQ был обречен рухнуть, как овца, вздумавшая поучиться летать.




*Ой, извините. Вы, наверное, покупали акции по индексу Wired?*

Но радость по поводу чужих несчастий не бывает долгой. Сейчас наступает уныние, и хотя я знаю, что *официально* у нас нет депрессии в экономике, но я чувствую себя угнетенно, не потому, что столько компаний так бездарно окончили свой путь, не успев его начать, а потому, что дух времени гнетет. И мы должны есть собачий корм, а не Krispy Kremes.

Что мы и делаем, потому что жизнь продолжается. И хотя все кругом ходят, низко опустив головы, оплакивая потерянные *часы*, здоровье и драгоценную жизнь, погубленные ради опциона акций в какой-нибудь SockPuppet.com, жизнь продолжается. И цикл производства продукта должен идти, и мы в Fog Creek вступаем в ту фазу цикла разработки продукта, когда должны есть корм собственных собак. Поэтому на время мы становимся Dog Creek Software.

Идиома to eat one's dog food (есть корм своей собаки) в компьютерной индустрии означает, что разработчик *использует* собственное ПО. Я совсем забыл, насколько это полезно, пока месяц назад не взял домой сборку CityDesk (предполагая, что до выпуска осталось недели три) и не попытался построить сайт с помощью этого пакета.

К моему удивлению, обнаружилось несколько ошибок, которые просто не позволили мне продолжить, и я вынужден был их исправить, прежде чем смог двинуться дальше. Все проведенное нами тестирование с дотош-

[1] Peter Schwartz (Питер Шварц) and Peter Leyden (Питер Лейден) «The Long Boom: A History of the Future, 1980–2020» (Затяжной бум: история будущего), *Wired*, July 1997. См. *www.wired.com/wired/archive/5.07/*.

ным раскрытием каждого пункта меню и проверкой правильности его работы не выявило тех обстоятельств, которые делали невозможным выполнение основной задачи продукта. Поработав с продуктом так, как это сделал бы покупатель, я обнаружил все эти «засады» за считанные минуты.

И это еще не все. По ходу работы – даже не обращаясь к специальным функциям, а просто попытавшись построить простой сайт – я обнаружил в то воскресное утро 45 ошибок. Я ленив и не мог потратить на это больше двух часов. И я не проверял ничего, кроме самой основной функциональности продукта.

Придя в понедельник на работу, я собрал всех и рассказал о 45 найденных ошибках. (По правде говоря, многие из них не были настоящими дефектами, просто некоторые функции можно было сделать более удобными.) Затем я предложил, чтобы каждый построил хотя бы один серьезный сайт с помощью CityDesk, чтобы выкурить и другие ошибки. Вот что значит быть производителем ПО, помещенным в шкуру пользователя.

Вот пример из собственного опыта.

Я предполагаю, что многие попытаются импортировать веб-страницы в CityDesk путем копирования и вставки кода HTML. Эта функция работает. Но когда я попытался импортировать реальную живую страницу из «New York Times», то потратил целый день, терпеливо редактируя HTML, отыскивая все ссылки IMG (на внешние картинки), загружая картинки из Интернета, импортируя их в CityDesk и настраивая ссылки IMG, чтобы они указывали на внутренние картинки. Трудно поверить, но одна статья на этом веб-сайте содержит 65 ссылок IMG на 35 различных изображений, часть из которых – всего лишь распорки шириной в 1 пиксел, которые очень трудно загрузить с помощью веб-броузера. А у CityDesk есть странная мания заменять имена импортированных изображений числами, а какими именно числами, он не сообщает, поэтому, не вдаваясь в детали, скажу, что импорт одной страницы в CityDesk отнял у меня целый день.

Это несколько расстроило меня, поэтому я вышел в сад и какое-то время занимался прополкой сорняков. (Просто не знаю, *как* мы будем снимать стресс, когда приведем сад в порядок. Хорошо еще, что мы не можем позволить себе нанять садовников.) И тут меня осенило. Я же программист в конце концов! За то время, что я импортировал одну страницу и настраивал для нее графику, я мог бы написать процедуру, которая делает это автоматически! Я управился даже *быстрее*. Теперь импорт страницы занимает примерно полминуты вместо одного дня и, в сущности, происходит без ошибок.

Здорово.

Вот зачем нужно «питаться собачьим кормом».

А когда Майкл попробовал сам импортировать некоторые сайты, он нашел штук 10 ошибок, которые ввел я. Например, обнаружились сайты, где у картинок сложные имена, которые не могут стать именами файлов при импорте, потому что в них есть вопросительные знаки, допустимые в URL, но не в именах файлов.

Иногда загрузишь программу и поражаешься тому, как она плоха или как трудно выполнить простейшие задачи, которые эта программа должна выполнять. Вполне вероятно, это из-за того, что разработчики программы не опробовали ее на себе.

А вот пример, в котором люди не хотят есть корм для своей собаки, и он еще интереснее. Как вы думаете, какой почтовой программой мы пользовались в Juno Online Services? (Если вы еще не знаете, то я проработал несколько лет, разрабатывая клиента Juno.)

Вы, наверное, думаете, что Juno? В конце концов, это ведь был *наш продукт*?

Нет. Пара человек, включая президента, установили Juno дома. Остальные 175 человек работали с Microsoft Outlook.

И на то были причины! Клиент Juno не был выдающимся почтовым клиентом; два года мы занимались только тем, что совершенствовали показ рекламы. Многие из нас считали, что если бы нас *вынудили использовать свой продукт*, нам пришлось бы его усовершенствовать, хотя бы для того, чтобы уменьшить свои мучения. Президент очень настаивал, чтобы мы показывали рекламу поочередно в шести разных местах, пока он не пришел домой, не получил шесть всплывающих рекламных объявлений и не сказал: «Пожалуй, двух было бы достаточно».

AOL увеличивала количество своих абонентов с сумасшедшей скоростью отчасти потому, что она предоставляла пользователю *более комфортные условия работы*, чем Juno, а мы этого не понимали, потому что не ели корм своей собаки. А не ели мы корм своей собаки потому, что он был отвратителен, а руководство действовало настолько неправильно, что нам было просто запрещено что-либо делать для того, чтобы сделать его хотя бы сносным на вкус.

Ну да ладно. CityDesk стал выглядеть *намного* лучше. Мы исправили все те ошибки, нашли другие и исправили их тоже. Мы добавляем пропущенные функции, необходимость которых стала очевидна. И мы приближаем-

ся к дню поставки! Ура! И, к счастью, нам больше не нужно соперничать с 37 компаниями, в каждой из которых 25 млн. венчурного капитала и которые борются с нами, раздавая свой продукт бесплатно тем, кто соглашается вытатуировать у себя на лбу огромное рекламное объявление. В экономике, пришедшей на смену «новой», все стараются посчитать, сколько они заработают, если будут брать плату. Если действовать с умом, то в этой наступившей экономике *можно жить*. Но все эти бесконечные сообщения о «дот-коме» свидетельствуют скорее об отсутствии изобретательности у редакторов бизнес-прессы, нежели о чем-то другом. Извините, fucked-company.com, это было интересно в течение месяца или около того, а теперь это выглядит жалко. Мы будем стараться улучшить наш продукт и остаться в этом бизнесе и для этого будем прислушиваться к нашим покупателям и есть корм нашей собаки, а не носиться по всей стране, пытаясь найти дополнительный венчурный капитал.

ГЛАВА ТРИДЦАТЬ ПЕРВАЯ

# Как делать дело, если вы всего лишь рядовой

25 ДЕКАБРЯ 2001 ГОДА, ВТОРНИК

Эта книга рассказывает об управлении разработкой программ. Но не каждый обладает такой властью, что может приказами менять что-либо в своей организации. Ясно, что если вы рядовой программист у основания тотемного столба, то не можете приказать начать составлять графики или вести базу данных ошибок. И даже если вы руководитель, то могли заметить, что управлять разработчиками – это все равно, что пасти котов, только скучнее. Если просто сказать «сделайте так», это не значит, что будет сделано как следует.

Работа в организации, характеризуемой низкими результатами по тесту Джоэла,[1] может быть причиной глубокой неудовлетворенности. Как бы хорош ни был ваш собственный код, код ваших коллег настолько плох, что вам стыдно участвовать в проекте. Или руководство принимает неудачные решения о том, какой код писать, вот и приходится тратить свой талант на отладку детской игры на тему пенсионного плана в версии для AS/400.

Наверно, вы могли бы уйти. Но предположим, что по какой-то причине вы там застряли. Не получили в полное владение опцион на акции, в вашей деревушке нет *лучше* места для работы, а может быть, ваш босс держит в заложниках объект вашей любви. В любом случае необходимость работы в слабой команде может действовать на нервы. Однако следуя некоторым стратегическим планам, можно улучшить команду «снизу», и я хочу поделиться некоторыми из них.

---

[1] См. главу 3.

## *Стратегия 1: просто делай*

Проект можно значительно улучшить стараниями одного лишь человека. Нет сервера для ежедневной сборки? Сделайте его. Настройте свою машину, чтобы она по расписанию осуществляла сборку ночью и посылала результаты электронной почтой. Слишком много действий нужно для сборки? Напишите makefile. Никто не проводит тесты юзабилити? Проводите собственные коридорные тесты юзабилити на рассыльных с листком бумаги и VB-прототипом.

## *Стратегия 2: воспользуйтесь силой вирусного маркетинга*

Любая из стратегий теста Джоэла может быть реализована одним-единственным человеком, причем команда в ней не участвует. Некоторые из этих стратегий, будучи осуществлены успешно, сами распространятся на остальных членов команды.

Допустим, что вы никого не можете убедить в пользе базы данных ошибок.[1] Ну и пусть. Ведите свою собственную. Заносите в нее ошибки, которые обнаружите в собственном коде. Обнаружив ошибку, которую должен исправить кто-то другой, запишите его ответственным за эту ошибку в своей базе данных. Если у вас хорошая программа слежения за ошибками, она пошлет им уведомление электронной почтой. *Продолжайте* посылать им сообщения, пока они не исправят свою ошибку. В конечном счете они увидят ценность системы и начнут ее применять. Если группа QA откажется вводить ошибки в систему учета ошибок, откажитесь принимать сообщения об ошибках по другим каналам. Когда вы в тысячный раз скажете им: «Рад бы это исправить, но я все равно забуду. Не могли бы вы ввести ошибку в систему?», они начнут использовать базу данных.

Никто не хочет применять систему контроля версий? Создайте собственное хранилище CVS, если нужно, то на своем диске. Даже в одиночестве вы можете заносить в нее свой код независимо от всех остальных. Когда у них возникнут проблемы, которые может решить система контроля версий (кто-то ненароком введет `rm * ~` вместо `rm *~`), они придут к вам за помощью. В конечном счете люди поймут, что могут загружать код из системы и себе.

---

[1] См. *www.joelonsoftware.com/articles/fog0000000029.html*.

## *Стратегия 3: создайте оазис качества*

Никто не хочет составлять графики?[1] Или спецификации?[2] Пишите свои собственные. Никто не станет возражать, если вы потратите день или два, чтобы написать минимальную спецификацию и график для работы, которую собираетесь сами сделать.

Постарайтесь, чтобы в команде появились хорошие специалисты. Участвуйте в поиске работников и в собеседованиях[3] и привлекайте хороших кандидатов.

Ищите тех, кто стремится к улучшениям и способен на них, привлекайте их на свою сторону. Даже в слабых командах встречаются толковые люди, у которых просто нет опыта создания хорошего кода. Помогите им. Настройте их на учебу. Читайте код, который они загружают. Если они сделают глупость, не посылайте им заносчивых писем с описанием их глупости. Это их только разозлит и заставит защищаться. Вместо этого с невинным видом зарегистрируйте ошибку, которая, как вы знаете, – результат их кода. Пусть они сами найдут ошибку, что будет для них гораздо полезнее.

## *Стратегия 4: нейтрализуйте дураков*

Даже в лучших командах попадаются один-два дурака. Самое неприятное, когда код этих плохих программистов нарушает работу вашего хорошего кода или хорошим программистам приходится тратить время на исправление кода плохих программистов.

Ваша цель как рядового пехотинца состоит в минимизации ущерба или в сдерживании. В какой-то момент один из этих гениев, потратив две недели, напишет такой дрянной код, который не будет работать никогда. Преодолейте свой соблазн потратить 15 минут и написать заново код, который будет работать правильно. У вас есть прекрасная возможность нейтрализовать этого недоумка на несколько месяцев. Просто регулярно сообщайте об ошибках в его коде. Ему ничего не останется, кроме как *месяцами* бороться с ними, пока вы не сможете уже больше отыскать ошибки. Все это время он не сможет наносить ущерб в каком-либо другом месте.

---

1 См. главу 9.

2 См. главу 5.

3 См. главу 20.

## *Стратегия 5: избавьтесь от отвлечений*

Все счастливые условия работы похожи друг на друга (закрытые офисы, тихая обстановка, отличные инструменты, редкие отвлечения и даже редкие большие совещания). Каждая несчастная рабочая обстановка несчастна по-своему.

Увы, изменить условия работы почти невозможно практически в любой компании. Долгосрочная аренда может означать, что даже руководитель фирмы не может ничего поделать. Поэтому очень немногие программисты работают в отдельных офисах. Это наносит ущерб их компаниям, по крайней мере, в двух аспектах. Во-первых, им труднее нанять первоклассных программистов, которые предпочтут фирму с более приятными условиями работы (при прочих равных). Во вторых, частота отвлечений резко снижает продуктивность разработчиков, которые не могут войти в ритм работы и сколько-нибудь долго в нем оставаться.

Поищите способ освободиться из этой среды. Возьмите свой лэптоп в кафетерий компании, где множество столиков пустует почти весь день (и где вас никто не найдет). Закажите конференц-зал на весь день и пишите код там, дав понять объемом сдаваемого в хранилище кода, насколько вы производительнее, когда работаете в комнате один. При очередном кризисе, когда начальник будет спрашивать, что вам нужно, чтобы «это было сделано к завтрашнему дню», вы уже будете знать, что ответить. Они найдут для вас помещение на день. А вскоре начнут размышлять, как сделать, чтобы такая продуктивность сохранялась в течение всего года.

Приходите на работу попозже и уходите попозже. Те часы, когда все остальные расходятся по домам, могут оказаться самыми продуктивными. Ну, конечно, если в вашей бригаде все регулярно приходят поздно, тогда приходите на работу к 9 утра. Вы сделаете больше за два часа до прихода остальных, которые начнут вам мешать, чем за весь оставшийся день.

Не держите постоянно в работе почтового клиента или IM. Проверяйте, если хотите, почту каждый час, но *не непрерывно*.

## *Стратегия 6: станьте незаменимым*

Ни одна из этих стратегий не будет действовать, если ваш вклад в работу не окажется действительно весомым. Если вы не будете писать много хорошего кода, то вас будут просто презирать за то, что вы суетитесь со всякими базами данных ошибок «вместо того», чтобы писать код. Нет ни-

чего вреднее для карьеры, чем репутация человека, так занятого процессом, что он не способен выполнять работу.

Однажды, поступив на работу рядовым программистом в некую компанию, я обнаружил, что она тянет примерно на 2 балла по тесту Джоэла, и решил это исправить. Но я также понимал, как важно произвести приятное первое впечатление. Поэтому каждый день я отводил первые семь часов написанию кода, как это от меня и ожидалось. Ничто не производит столь хорошего впечатления на остальных членов команды, как шквал загрузок кода в хранилище. Но перед тем как идти домой, я каждый день отводил один час совершенствованию технологии. В это время я исправлял то, что мешало отлаживать наш продукт. Я организовал ежедневную сборку и базу данных ошибок. Я убрал все давнишние досадные неприятности, затруднявшие разработку. Я написал спецификации для работы, которой занимался в течение дня. Я составил документ, где шаг за шагом разъяснялось, как с самого начала создать механизм совершенствования. Я тщательно задокументировал важный внутренний язык, который до того не был документирован. Постепенно технология становилась все лучше. Никто кроме меня (и моей команды, когда я стал ею руководить) никогда не составлял графики или спецификации, но в остальном мы достигли примерно 10 баллов по тесту Джоэла.

Вы *можете* улучшить положение, даже не будучи начальником, но вы должны быть вне подозрений – как жена Цезаря. В противном случае вы наживете себе врагов.

ГЛАВА ТРИДЦАТЬ ВТОРАЯ

# Две истории

*В конце 1999 года Дейв Винер (DaveWiner) открыл онлайновую службу под названием EditThisPage.com (РедактируйЭтуСтраницу.com) и предложил людям создавать сайты с помощью того же формата веблога, который он установил для собственного сайта Scripting News. Так получилось, что в то же самое время я был совершенно разочарован своей работой в компании, где большие таланты тратились впустую из-за того, что стиль управления напоминал тактику набегов (расшифровываю: за краткими периодами мелочной опеки следовали долгие периоды полного забвения). В результате я написал классическую проповедь Джоэла в стиле традиций блоггеров: рассказ о собственных несчастьях в виде рекомендации по менеджменту.*

*К счастью, в те дни меня никто не читал, и мне сошли с рук указание реальных имен и даже риск оскорбить миллиардеров Сиэтла, которые ездят на машинах, стоящих дороже, чем все мои банковские счета и пенсионные накопления, вместе взятые. Да, золотые были дни.*

19 МАРТА 2000 ГОДА, ВОСКРЕСЕНЬЕ

Хочу рассказать две истории из моей карьеры, которые считаю классическими иллюстрациями разницы между компаниями, где управление поставлено на здоровую основу, и компаниями, обреченными на катастрофу. По сути это разница между доверием к работникам, имеющим право делать то, что нужно, и недоверием к ним как к недотепам, каждый шаг которых необходимо контролировать, иначе эти саботажники все сорвут.

Мое первое задание на первой работе относится к Microsoft, где от меня потребовали предложить новую стратегию макроязыка для Excel. В скором времени у меня был готов проект спецификации Excel Basic (позднее развившегося в Visual Basic for Applications, но это другая история). Каким-то образом таинственная группа людей в Microsoft, называвшаяся «Группой архитектуры приложений», пронюхала про мою спецификацию, которая, должно быть, озадачила их, потому что по каким-то причинам им показалось, что *они* отвечают за такие вещи, как стратегии макроязыков, и они пожелали увидеть мою спецификацию.

Я в свою очередь заинтересовался. Что это еще за группа «Архитектуры приложений»? Похоже, что никто не воспринимал их всерьез. Это была группа из четырех недавно нанятых человек со степенью Ph.D (что для Microsoft весьма необычно). Я послал им экземпляр моей спецификации и отправился на личную встречу в надежде, что они могут сказать что-либо интересное.

«Ля-ля-ля!» – сказал один из них. «Ля-ля-ля, ля-ля-ля!» – сказал другой. По-моему, ничего интересного от них услышать было нельзя. Они были захвачены идеей создания подклассов и, похоже, считали, что тем, кто станет создавать в Excel макросы, понадобится много подклассов. Во всяком случае один из них произнес: «Хорошо, это все очень интересно. Что дальше? Кто будет утверждать вашу спецификацию?»

Я рассмеялся. Хотя я проработал в Microsoft всего несколько месяцев, мне уже было ясно, что понятия «*утверждать* спецификацию» здесь не существует. Да тут ни у кого нет времени на то, чтобы просто *прочесть* ее, не то, что утвердить. Программисты дергали меня ежедневно, чтобы я дал им новые страницы и они могли писать код дальше. Мой босс (и его босс) дали мне ясно понять, что никто не понимает макросов и не располагает временем, чтобы работать над ними, поэтому, что бы я ни делал, это будет под мою ответственность. А тут этот Ph.D из странной исследовательской группы считает, что должны быть какие-то формальности.

Довольно скоро я понял, что группа архитектуры приложений знает о макросах еще меньше меня. Я хотя бы поговорил с несколькими разработчиками макросов и ветеранами работы над Excel, чтобы уяснить, какие реальные задачи выполняют люди с помощью макросов Excel: пересчитывают каждый день таблицу или реорганизуют данные по какой-нибудь схеме. А группа архитектуры приложений лишь *рассуждала* о макросах в качестве некоего научного упражнения, и предложить примеры макросов, которые захотят писать люди, они не могли. Под моим давлением

один из них выдвинул идею, что поскольку в Excel уже есть подчеркивание одной чертой и подчеркивание двумя чертами, то кто-нибудь захочет написать макрос для подчеркивания тремя чертами. Да. Очень актуальная задача. В общем, я постарался впредь игнорировать их, проявляя как можно большую дипломатичность.

Похоже, что это разозлило человека по имени Грег Уиттен, возглавлявшего группу архитектуры приложений. Надо сказать, что Грег числился в Microsoft примерно 6-м номером по влиянию. Он там работал с незапамятных пор. Никто не мог вспомнить что-нибудь сделанное им, но, по-видимому, он часто обедал с Биллом Гейтсом, и GW-BASIC был назван в его честь. Грег созвал большой хурал и стал там жаловаться, что бригада Excel (имея в виду меня) заваливает стратегию макросов. Мы потребовали, чтобы он привел конкретные обоснования, но его аргументы были неубедительны. Надо же, подумал я, никому не известный, только что принятый на работу сразу после колледжа – и спорит с человеком номер 6, очевидно, одерживая верх. (Можете представить себе такое в компании серых фланелевых костюмов?) Моя бригада программистов, возглавлявшаяся Беном Уолдменом (сейчас он вице-президент Microsoft), полностью меня поддержала, что было самым важным, поскольку программисты писали код и давали окончательную оценку того, что делалось.

Я вполне удовлетворился бы, останься все, как было. Если группа архитектуры приложений требует внимания и заботы и хочет что-то пообсуждать, я не возражаю и буду участвовать с ними в обсуждении сколько угодно при условии, что они оставят в покое программистов и дадут им возможность работать. Но затем произошло нечто еще более интересное и поразившее меня. Мы сидели и завтракали с несколькими коллегами, под редмондским солнышком, когда ко мне подошел Пит Хиггинс. В то время Пит был генеральным руководителем проекта Office – я, конечно, знал, кто он, но не ожидал, что он хорошо осведомлен об этой истории.

«Как дела, Джоэл? – спросил он меня. – Я слышал, у тебя какие-то проблемы с группой архитектуры приложений?»

«Нет! – сказал я. – Во всяком случае я сам с ними справляюсь».

«Можешь не продолжать, – сказал он, – я понял». И ушел. На следующий день до меня дошел слух: группа архитектуры приложений расформирована. Более того, все ее члены отправлены в разные подразделения Microsoft, как можно дальше друг от друга. Больше я о них не слышал.

Я, конечно, был потрясен. В Microsoft, если вы менеджер программы и работаете над стратегией макросов Excel, пусть даже находясь в компа-

нии меньше полугода, вы – БОГ стратегии макросов Excel, и никому, даже человеку номер 6, не позволено вам мешать. Точка.

В этом заложен глубокий смысл. Во-первых, это заставляет каждого гораздо добросовестнее относиться к своей работе. Нельзя спрятаться за то, что «руководство одобрило мою спецификацию», потому что руководство не особенно пристально изучало твою спецификацию. Руководство сделало свое дело: наняло толковых людей и поставило перед ними задачу. С другой стороны, работать в таких условиях чрезвычайно приятно. Кто же не хочет царствовать в своих владениях? Программное обеспечение по природе своей допускает разделение на малые компоненты, поэтому всегда можно поделить между людьми ответственность и дать им возможность *владеть* своей областью. Возможно, это главная причина, по которой программисты любят работать в Microsoft.

Прошли годы. Я оказался на работе в компании Juno, предоставлявшей онлайновые услуги и бесплатную электронную почту. Условия работы там были совсем не такие, как в Microsoft. В моем подчинении было два программиста, но мой начальник постоянно подрывал мою (ограниченную) власть тем, что напрямую обращался к моим подчиненным и давал им задания, часто даже не сообщая мне об этом. Даже такой тривиальный вопрос, как предоставить работнику выходной день или нет, мой руководитель считал своей прерогативой.

Через пару лет после прихода в Juno я работал над новой функцией для регистрации абонентов. В Juno 3.x, основной версии, я должен был отвечать за полную переработку процедуры регистрации. К тому времени я был относительно старшим членом в группе технического обеспечения. Я получал отличные оценки личной продуктивности, и моим руководителям, похоже, нравилось то, что я делаю. Но они никак не могли преодолеть недоверие ко мне. Отдавай приказы и проверяй исполнение.

В одном месте процедуры регистрации пользователю предлагалось ввести дату своего рождения. Это была лишь малая часть долгой процедуры регистрации, занимавшей около 30 экранов, в течение которой Juno допрашивала вас о ваших доходах, любимых видах спорта, количестве детей и их возрасте и сотне других вещей. Чтобы немного облегчить пользователю процедуру регистрации, я хотел изменить поле даты рождения и установить для него свободный формат, чтобы можно было вводить «8/12/74», или «August 12, 1974», или «12 Aug 74», или как-нибудь еще. (Вы работали в Outlook? Это должно было быть, как в Outlook, где можно вводить даты практически в любом формате и они принимаются.)

Не особенно разобравшись, мой менеджер решил, что ему это не нравится. Для него это стало предметом самоутверждения. Сперва он наорал на дизайнера, работавшего над этой страницей (даже не сообщив мне). Потом он наорал на меня. Потом он стал каждый день напоминать мне, что я должен исправить и сделать так, как *ему* хочется. Затем он привел главу компании, чтобы продемонстрировать мою работу, и устроил большое шоу, добившись критики моего нового дизайна главой компании. В Juno даже глава фирмы любит вмешиваться в работу, выполняемую на самом нижнем уровне, т. е. там это обычное явление.

Нетрудно догадаться, что я пришел в ярость. Мелочь, по сути, дело вкуса. Одним больше понравится мой вариант. Другим больше понравится его. Но смысл был ясен: здесь ты *будешь* делать так, как тебе сказано. Такой стиль руководства скорее позволяет выяснить, у кого крепче нервы, чем обсуждать дизайн интерфейса пользователя.

Не скажу, что ушел из Juno именно поэтому, но описанный случай иллюстрирует причину моего ухода. Стало понятно, что неважно, как ты стараешься, неважно, насколько ты умен, неважно, есть ли у тебя чувство ответственности за что-нибудь или нет, ты не управляешь ничем, даже самыми мелкими мелочами. Ничем. Все свои чертовы идеи, образование, ум, сообразительность – все, за что тебе платят – можешь засунуть в одно место. А в Juno было *множество* администраторов, наверно, четверть всех служащих, и они постоянно совались во все решения, чтобы показать, что командуют тут *они*. Разительный контраст с Microsoft, где вице-президент спускается со Здания 9, чтобы подтвердить, что у тебя есть все полномочия для выполнения задачи.

В какой-то мере безнадежная некомпетентность руководства Juno объясняется тем, что эта компания расположена в Нью-Йорке, а не на западном побережье, и современные методы управления плохо проникли в нее. Эта проблема вызвана также крайней неопытностью менеджеров Juno, начиная с самого верха – 29-летнего главы, который никогда не работал нигде, кроме D.E. Shaw, который вмешивается во все, что только ему попадается, включая текст сообщений об ошибках, показываемых при сбоях; главный технический специалист регулярно кричит на своих подчиненных, если они осмеливаются оспаривать его мудрость; те отыгрываются на программистах, которые, в свою очередь, приходят домой и пинают своих собак. И сравните это с Microsoft, где все делается на самом низком уровне, а большинство менеджеров ведут себя так, будто главная их задача – передвинуть мебель в комнате так, чтобы она не мешала и программисты могли сосредоточиться на своей работе.

ГЛАВА ТРИДЦАТЬ ТРЕТЬЯ

# Биг-Маки против «Голого повара»

18 января 2001 года, четверг

Загадка: почему некоторые из самых крупных в мире компаний, занимающихся ИТ-консалтингом, работают хуже всех?

Почему новые блистательные консалтинговые компании начинают с феерических успехов, растут со скоростью метеоров и быстро скатываются в болото посредственности?

Думая об этом, я думаю и о том, как должна развиваться моя собственная компания. И лучшие примеры я нахожу у Макдональдс. Да, я имею в виду эту ужасную сеть торговли гамбургерами.

Секрет биг-маков в том, что все они не бог весть что, но каждый из них «не бог весть что» в равной степени. Если вас устраивает не бог весть что, то покупайте биг-мак, и никакое удивление вам не грозит.

Другой секрет биг-маков состоит в том, что IQ повара может находиться в диапазоне между «идиотом» и «слабоумным» (это легальные термины), но он все равно способен изготавливать биг-маки, которые ничем не отличаются от всех остальных биг-маков в мире. Дело в том, что *настоящий* рецепт соуса Макдональдс – это его огромная техническая инструкция, в которой подробнейшим образом описывается, какие действия должны выполняться в каждом предприятии сети при изготовлении биг-мака. Если гамбургер биг-мак жарится за 37 секунд в Анкоридже, штат Аляска, то он будет жариться за 37 секунд и в Сингапуре, а не за 36 и не за 38. Чтобы сделать биг-мак, вы просто должны выполнять эту чертову инструкцию.

Инструкции были тщательно разработаны весьма неглупыми людьми (из Университета Макдональдс Гамбургер[1]) с тем, чтобы их мог выполнять как болван, так и умный человек. Эти инструкции содержат все меры предосторожности (например, сигнал, который подается, если что-то жарится в масле слишком долго), призванные компенсировать недостатки человеческого фактора. Секундомеры и системы учета времени стоят всюду. Одна из этих систем смотрит, проверяют ли уборщики чистоту туалетных комнат каждые полчаса. (Подсказка: не проверяют.)

Система предполагает, что каждый наделает кучу ошибок, но гамбургеры получатся такими, как надо, и у вас всегда спросят, хотите ли вы к ним жареной картошки.

В порядке развлечения сравним повара из Макдональдс, который в точности следует инструкции и ничего не понимает в еде, с таким гением, как «голый повар», умница-британец Джейми Оливер. (Если вы сейчас предпочтете закрыть книгу и отправиться на его веб-сайт,[2] чтобы посмотреть видео в стиле MTV о том, как надо готовить айоли с базиликом, мое вам благословение. Идите на здоровье.) Конечно, сравнивать Макдональдс и повара для гурманов – полный абсурд, но придержите на время свое недоверие, потому что здесь есть кое-что поучительное.

Во-первых, голый повар не следует никаким производственным инструкциям. Он не отмеряет *ничего*. Пока он готовит, какие-то продукты летают по кухне. «Добавим сюда чуточку розмарина, это не помешает, и хорошенько все встряхнем, – говорит он. – Перемешаем. Отлично. Теперь просто раскидаем это по тарелке». (Да, и вы видите, как он просто раскидывает все по тарелке. Конечно, если бы *я* попытался раскидать все по тарелке, у меня ничего не получилось бы.) За 14 секунд он сымпровизировал деликатес из жареного филе морского окуня, фаршированного овощами, запеченного на шампиньонах с соусом сальса-верде.

Я думаю, достаточно очевидно, что блюда голого повара лучше, чем гастрономические изыски от Макдональдс. Хотя это звучит глупо, но стоит на минуту задуматься *почему*. Это не такой глупый вопрос. Почему крупная компания с неизмеримыми ресурсами, огромными масштабами, возможностью привлечения лучших гастрономов, каких только можно купить за деньги, и неистощимыми финансами не может производить отличную еду?

---

[1] Makdonald's Hamburger University – корпоративный университет компании McDonald (университет «имени булки с котлетой»), основан в 1961 г. – *Примеч. ред.*

[2] См. *www.jamieoliver.net/*.

Представьте себе, что голому повару надоест ТВ и он откроет ресторан. Конечно, он великолепный повар, он приготовит изысканные блюда, и от посетителей отбоя не будет, а заведение станет высокодоходным.

Когда у вас будет высокодоходный ресторан, вы быстро поймете, что, даже если брать 19 долларов за закуску и 3,95 доллара за колу, ваши доходы достигнут естественного предела, поскольку один повар может приготовить ограниченное количество пищи. Поэтому вы наймете еще одного повара и, возможно, откроете филиалы (может быть, в других городах).

Тут-то вы и столкнетесь с тем, что в технике называют *проблемой масштабирования*. Пытаясь клонировать ресторан, вы должны выбирать между наймом другого выдающегося повара необходимого калибра (но тогда этот повар захочет, вероятно, оставить себе значительную часть той прибыли, которую он создал, и нет смысла стараться) либо наймом повара подешевле и помоложе, который не будет столь умел, в чем ваши клиенты вскоре убедятся и перестанут ходить в новый ресторан.

Стандартный способ решения проблемы масштабируемости – нанять дешевых поваров, которые ничего не умеют, и дать им такие точные инструкции приготовления каждого блюда, чтобы они «не могли» его испортить. Просто выполняй эти правила, и ты будешь делать великолепные блюда для гурманов!

Проблема: на практике все происходит несколько иначе. Очень многое из того, что делает хороший повар, основано на *импровизации*. Хороший повар видит на рынке какие-нибудь необычайные манго и импровизирует сальсу с манго и кинзой для рыбного блюда дня. Хороший повар, у которого вдруг кончится картофель, временно заменит его каким-нибудь таро. Повар автоматизированного производства сможет приготовить данное блюдо, только если все работает идеально, но без настоящего таланта и мастерства он не сможет импровизировать, поэтому вы никогда не увидите хикаму в Макдональдсе.

Картофель в Макдональдсе совершенно *определенного* сорта, они выращивают его по всему свету, заранее режут и замораживают в огромных количествах на случай перебоев в поставках. Из-за того, что картофель режут и замораживают, картофель фри получается не таким вкусным, но он *соответствует* стандарту и не требует мастерства от повара. Макдональдсом предусмотрены сотни процедур, обеспечивающих изготовление продукта удовлетворительного качества *любым слабоумным, который окажется на кухне*, даже если это качество «немного» ниже, чем могло бы быть.

Подведем итог сказанному:

1. Чтобы действительно хорошо делать некоторые вещи, нужен талант.
2. Талант трудно масштабировать.
3. Один из способов масштабирования таланта – предложить его носителю написать инструкции, которые должны выполнять те, у кого таланта нет.
4. Качество получаемого продукта очень низкое.

*Абсолютно такая же ситуация* имеет место в ИТ-консалтинге. Думаю, вы неоднократно слышали историю, которую я сейчас расскажу.

Майк был расстроен. Он нанял огромную компанию ИТ-консультантов, чтобы построить Систему. Нанятые им ИТ-консультанты были некомпетентны, все время говорили о «Методологии», потратили миллионы долларов и ничего не сделали.

К счастью, Майку попался молодой программист, действительно толковый и талантливый. Юный программист создал всю систему в течение одного дня за 20 долларов и пиццу. Майк пришел в полный восторг и рекомендовал юное дарование всем своим друзьям.

Юный программист начал грести деньги лопатой. Скоро у него было больше заказов, чем он мог выполнить, поэтому он нанял себе помощников. У хороших программистов слишком хороший аппетит на участие в капитале компании, поэтому он решил нанимать еще более молодых программистов прямо после колледжа и «обучать их» в течение шести недель.

Беда в том, что результаты этого «обучения» оказываются очень неровными, поэтому Юный Программист начинает составлять правила и процедуры, призванные обеспечить более стабильные результаты. С годами сборник инструкций становится все толще и толще. Вскоре он становится уже руководством в шести томах под названием «Методология».

Через пару десятков лет Юный Программист превратился в Крупного Некомпетентного ИТ-консультанта, обладающего методологией с заглавной буквы и управляющего множеством людей, слепо повинующихся этой Методологии, даже если она не приносит результатов, потому что у них нет ни малейшего понятия о том, что еще можно сделать, и потому что у них нет таланта к программированию у них общетехническое образование и шесть недель курсов за плечами.

И у Нового Крупного Некомпетентного ИТ-консультанта возникают большие трудности. Клиенты недовольны. Приходит другой юный талантливый программист, отнимает у него весь бизнес, и все повторяется сначала.

Нет необходимости называть имена; этот цикл повторялся десятки раз. Все компании, предоставляющие услуги в ИТ, становятся жадными и пытаются расти быстрее, чем могут находить талантливых людей, и начинают громоздить одни правила и процедуры поверх других, чтобы обеспечить «стабильное», если уж не блестящее, качество работы.

Но правила и процедуры эффективны, пока ничего не случится. За последнюю пару лет появилось множество консалтинговых компаний по созданию «веб-сайтов, управляемых данными», набиравших сотрудников из рядовых любителей, которым рассказывали о четырнадцати вещах, необходимых для создания управляемого данными веб-сайта («вот тебе, парень оператор select, строй веб-сайт»). Но доткомы полопались, и вдруг возник спрос на программирование клиентского GUI, владение C++ и настоящее знание информатики. Для мальчиков, имеющих в своем арсенале только оператор select, кривая обучения оказывается слишком крутой, и они не успевают. Но они все равно пытаются использовать правила Книги III, главы 17, параграфа 29.4b, где говорится о нормализации баз данных, но почему-то эти правила не удается применить к Новому миру. Блестящие *основатели* этих компаний могут, конечно, приспособиться к новому миру: это талантливые специалисты в компьютерных технологиях, которые могут освоить что угодно, но *компания, которую они создали*, не может приспособиться, потому что в ней талант заменен сборником инструкций, а сборники инструкций не могут приспосабливаться к новым временам.

Какова же мораль этого рассказа? *Берегись Методологий.* Они прекрасно обеспечивают жалкое, но терпимое качество труда каждого, но они раздражают людей талантливых, укладывая их в прокрустово ложе ограничений. Талантливый повар, конечно, не будет счастлив, испекая бургеры в Макдональдсе, именно *из-за* инструкций, насаждаемых Макдональдсом. Почему же ИТ-консультанты так хвалятся своими методологиями? (Не могу понять.)

А что же моя компания, Fog Creek? Положим, мы никогда не ставили себе цель стать крупной консалтинговой компанией. Для нас консалтинг – это вспомогательная деятельность на пути к долгосрочной цели – стать софтверной компанией, приносящей постоянную прибыль. И мы достигли этого, дополнив прибыль от продажи ПО доходами от некоторых консалтинговых работ. За какую-то пару лет наши доходы от ПО выросли настолько, что консалтинг стал лишь малоприбыльным отвлечением, поэтому сейчас мы беремся за него только тогда, когда это напрямую связано с поддержкой наших программных продуктов. ПО, как вы знаете, масшта-

бируется *чрезвычайно* хорошо. Когда кто-то еще покупает FogBUGZ, мы зарабатываем деньги, ничего при этом не потратив.

Еще важнее, что мы стремимся нанимать самых лучших... Нас полностью устраивает, что компания не растет из-за того, что нам не удается найти достаточное количество работников (хотя может показаться, что при шести неделях ежегодного отпуска с поиском людей не должно быть никаких проблем). И пока те, кого мы уже взяли, не усовершенствуют свои знания настолько, чтобы быть учителями и наставниками для новичков, мы не станем расширяться.

ГЛАВА ТРИДЦАТЬ ЧЕТВЕРТАЯ

# Все не так просто, как может показаться

*Общий термин «экстремальное программирование» объединяет массу замечательных идей, несколько средних и одну поистине опасную: что планирование и проектирование – это пустая трата времени. Философия экстремального программирования пропагандирует некую законченную методологию разработки, кажущуюся разумной, но на практике программисты часто прикрываются этой философией, чтобы сначала реализовывать функции, а потом их проектировать. «Исходный код – это и есть проект!» – говорят они. Но это не так, и если вы станете разрабатывать программы таким способом, то будете бесконечно толочь воду в ступе, отыскивая в программе «душок» и пытаясь его устранить, вместо того чтобы двигаться вперед.*

4 МАРТА 2002 ГОДА, ПОНЕДЕЛЬНИК

В CityDesk возникла небольшая проблема с юзабилити.

Вот в чем она состояла. Можно было импортировать файлы из Интернета с помощью команды меню (`Import Web Page`). А можно было импортировать в CityDesk файлы с диска, перетаскивая их мышью. И люди либо не замечали, что это можно сделать, либо пытались импортировать с диска с помощью функции `Import Web Page`, которая не работала так, как им хотелось.

Я решил, что это легко исправить с помощью мастера на двух страницах. Грубо говоря, первая страница будет спрашивать, откуда вы хотите импортировать файл, и если вы выбираете диск, то следующая страница предлагает выбрать файл, а если вы выбираете Интернет, то вторая страница предлагает указать URL.

Я уже почти принялся за реализацию, но что-то меня остановило, и вместо этого я написал мини-спецификацию. Вот эта спецификация в полном виде:

> **Страница 1**
> Откуда вы хотите выполнить импорт? (диск/веб)
>
> **Страница 2 (диск)**
> Стандартный диалог File/Open
>
> **Страница 2 (веб)**
> Мини-броузер с приглашением ввести URL

Внезапно я задумался. А можно ли встроить в мастер стандартный диалог Windows, прототип которого обычно предоставляется ОС?

Гм. Пришлось изучить этот вопрос. Оказалось, что можно, но дело это не простое и отнимет несколько часов.[1] Нельзя ли обойтись без мастера? Я переделал спецификацию:

> **Два пункта меню:**
>
> 1. **Импортировать веб-страницу из Интернета** → Показывает диалоговое окно URL.
> 2. **Импортировать веб-страницу с диска** → Показывает диалог открытия файла.

Гораздо лучше. Три минуты проектирования, и я высвободил несколько часов, которые ушли бы на написание кода.

Если вы хотя бы 20 минут в своей жизни писали код, то, вероятно, открыли для себя хорошее эмпирическое правило: *все не так просто, как может показаться*.

Такая простая вещь, как копирование файлов, таит массу опасностей. Что если первый аргумент – это каталог? Что если второй аргумент – файл? Что если файл с таким именем, как у копируемого, уже существует? Что если нет прав на запись?

Что если в середине копирования произойдет сбой? Что если приемник находится на удаленной машине, которая доступна, но требует аутентификации? Что если файлы большие, а канал узкий, и нужен индикатор выпол-

[1] См. *www.vbaccelerator.com/codelib/cmdlgd/cmdlgtp.htm*.

нения копирования? Что если скорость передачи упадет почти до нуля… в каком случае отказаться от передачи и возвратить сообщение об ошибке?

Хороший способ проверить кандидатов в тестеры – это дать им простую операцию и попросить перечислить все неприятности, которые, по их мнению, могут в ней произойти. Классический вопрос на интервью для тестеров в Microsoft: как бы вы стали тестировать диалоговое окно открытия файла? Хороший тестер выложит несколько десятков необычных ситуаций, которые нужно проверить («другой пользователь удалил файл в промежутке между тем моментом, когда он был показан в окне, и тем моментом, когда вы его выделили и нажали кнопку „открыть“»).

Итак, у нас есть одна аксиома: все не так просто, как может показаться.

В программировании действует и другая аксиома: всегда старайтесь снизить риск. Одним из наиболее важных видов риска связан с графиком работ, когда какая-то работа отнимает больше времени, чем вы предполагали. Это плохо, потому что начальник станет кричать на вас, а это неприятно. Если такой мотивации вам недостаточно, то подумайте об экономических факторах. Например, вы решили реализовать какую-то функцию, исходя из предположения, что для этого нужна одна неделя. Потратив вместо одной недели двадцать, вы поняли, что скорее всего приняли ошибочное решение. Чем больше таких неверных решений вы примете, тем выше вероятность, что большие сумки с логотипом вашей компании окажутся на складе ликвидатора, а ваш бывший президент будет скулить, что успех был таким недолгим, что о закрытии компании даже не сообщили на *fuckedcompany.com*.

Сочетание аксиом «все не так просто, как может показаться» и «сокращай риски» приводит к единственно верному выводу:

**Прежде чем что-то реализовывать, это что-то надо проектировать.**

Жаль, если я вас разочаровал. Да, я знаю, вы читали Кента Бека и теперь думаете, что можно не проектировать перед тем, как реализовывать. Извините, нельзя. *Нельзя* изменять код «с такой же легкостью», с какой можно изменять проектную документацию. Очень часто повторяют ошибочное утверждение: «Мы сейчас применяем инструменты такого высокого уровня, как Java и XML. Мы можем изменить код за считанные минуты. Почему же не проектировать в коде?» Дорогой друг, ты можешь приделать колеса своей маме, но она от этого не станет автобусом, и если ты считаешь, что можешь переработать свою плохо реализованную функцию копирования файлов так, чтобы она стала вытесняющей, а не выполнялась в отдельном потоке, и так же быстро, как я написал это предложение, то ты слишком упрям.

Во всяком случае я не считаю, что экстремальное программирование действительно отрицает проектирование. Речь идет о том, что «не надо проектировать больше, чем это необходимо», и это правильно. Но люди слышат другое. Большинство программистов ищут любые оправдания, чтобы не делать самого элементарного проекта, прежде чем реализовывать функции. Поэтому идея «никаких проектов» манит их, как горящая свеча манит мотыльков (путь которых заканчивается звуком «пшшик»!). Это одна из тех странных форм лени, когда в конечном итоге приходится выполнить работы даже больше. Я слишком ленив, чтобы сначала рисовать проект функции на бумаге, поэтому я напишу какой-нибудь код, а если он окажется неверным и его придется исправлять, то я потрачу больше времени, чем если бы я поступил иначе. Или, что бывает чаще, я напишу какой-нибудь код, и он окажется неверным, но времени нет, и мой продукт слаб, и оставшееся время я буду сочинять объяснения, почему это «должно быть так». Это просто нечистоплотно и непрофессионально.

Когда Линус Торвальдс атаковал проектирование,[1] он говорил об очень больших системах, которые должны развиваться, чтобы не превратиться в Multics. Он имел в виду не ваш код для копирования файлов. А если вы учтете, что у него была весьма четкая дорожная карта, по которой было ясно видно, куда идти, то неудивительно, что Линус не видит особого смысла в проектировании. Но не обольщайтесь. Скорее всего, к вам это не относится. А кроме того, Линус гораздо умнее нас, поэтому то, что хорошо для него, может не подойти нам, обычным людям.

Инкрементное проектирование и реализация – это хорошо. Частые релизы – это прекрасно (хотя если это касается коробочных или массовых продуктов, то покупатели сойдут с ума, чего лучше не допускать, поэтому надо почаще собирать внутренние версии). Слишком большой формализм в проектировании – пустая трата времени: никогда не видел, чтобы проект выиграл от бездумного вычерчивания диаграмм UML, CRC или чего-то еще, что может оказаться в большой моде. А все эти системы-монстры в 10 миллионов строк кода, о которых говорил Линус, должны эволюционировать, потому что люди толком не знают, как проектировать программы такого масштаба.

Но если надо написать функцию копирования файлов или запланировать функции для следующей версии продукта, то без проекта не обойтись. И пусть никакие сирены не увлекут вас в другую сторону.

---

[1] См. *wwwwsg.iu.edu/hypermail/linux/kernel/0112.0/0004.html*.

ГЛАВА ТРИДЦАТЬ ПЯТАЯ

# В защиту синдрома «это придумали не здесь»

14 ОКТЯБРЯ 2001 ГОДА, ВОСКРЕСЕНЬЕ

Проведем краткую викторину.

1. Повторно использовать код
   a) Хорошо
   b) Плохо
2. Изобретать колесо заново
   a) Хорошо
   b) Плохо
3. Синдром «это изобрели не здесь»
   a) Хорошо
   b) Плохо

Конечно, *все знают*, что созданное другими надо использовать всегда. Правильные ответы, разумеется: 1(a), 2(b) и 3(b).

Верно?

Не так быстро!

Синдром «это изобрели не здесь» считается классической патологией менеджмента и состоит в том, что команда отказывается пользоваться технологией, которую создала не она сама. Очевидно, что люди с таким синдромом просто мелочны, не хотят делать того, что выгодно всей организации, потому что им это не принесет никакой славы. (Верно?) В отделе нудных исторических исследований бизнеса вашего местного книжного мегамагазина полно рассказов о бездарных командах, потративших миллионы долларов и десяток лет, чтобы сделать то, что можно было купить в CompUSA за $9,99. И каждый, кто в течение трех последних десятилетий обращал

*хоть какое-то* внимание на прогресс в программировании, знает, что повторное использование – это Священный Грааль всех современных систем программирования.

Верно. Я тоже так считал. Поэтому, когда я был руководителем программы и отвечал за первую реализацию Visual Basic for Applications, я старательно собрал коалицию из четырех – отметьте количество! – разных бригад в Microsoft для получения пользовательских диалогов в Excel VBA. Идея была сложной и чревата взаимозависимостями. Существовала бригада под названием AFX, работавшая над созданием своего рода редактора диалогов. Потом мы должны были использовать совершенно новый код из группы OLE, позволявший встраивать одно приложение в другое. А команда Visual Basic должна была предоставить язык программирования, который бы все это обеспечивал. После недели переговоров мне удалось склонить бригады AFX, OLE и VB к принципиальному согласию.

Я зашел в кабинет Эндрю Кватинца (Andrew Kwatinetz). В то время он был моим менеджером и научил меня всему, что я знаю. «Бригада Excel никогда на это не пойдет, – сказал он. – Знаешь их девиз? «Найди зависимости – и избавься от них». Они никогда не согласятся с тем, в чем так много зависимостей».

Ин-те-рес-но. Я этого не знал. Пожалуй, это объясняет, почему у бригады Excel был свой компилятор C.

Конечно, многие мои читатели сейчас катаются на полу от хохота. «Ну и дураки там в Microsoft, – думают они. – Отказались использовать чужой код, да еще завели собственный компилятор всего для одного продукта».

Полегче, полегче! Сильное стремление бригады Excel к независимости привело также к тому, что они всегда выпускали продукт вовремя, и их код неизменно был высокого качества, и у них был компилятор, который, еще в 1980-х, генерировал *p*-код и потому мог выполняться без модификации как на чипе 68000 Macintosh, так и на Intel PC. Кроме того, *p*-код давал исполняемые файлы вдвое меньшего размера, чем двоичные файлы для Intel, поэтому он быстрее загружался с дискет и требовал меньше памяти.

«Найди зависимости – и избавься от них». Когда вы работаете в по-настоящему замечательной бригаде с отличными программистами, то, честно говоря, чужой код оказывается полным багов мусором, и никто, кроме вас, не умеет вовремя выдавать продукт. Если вы – шеф-повар с голубой лентой, и вам *нужна* свежая лаванда, то вы вырастите ее сами, а не станете покупать на рынке, потому что там может не оказаться свежей лаванды

или там окажется только старая лаванда, которую вам попытаются сбыть как свежую.

И в самом деле, во время последней дотком-мании какие-то шарлатаны, пишущие на темы, связанные с бизнесом, предположили, что компании будущего станут совершенно виртуальными: парочка стильных людей будет прихлебывать Шардонне у себя дома, а все практические операции будут осуществляться какими-то подрядчиками. Эти опьяневшие «провидцы» упустили из виду то, что рынок платит за добавленную стоимость. Два юппи в гостиной, которые покупают движок для электронной коммерции у компании A, продают товары, изготовленные компанией B, хранящиеся на складе и доставляемые покупателям компанией C, и поручают обслуживание клиентов компании D, добавляют не так много стоимости. Если вам случалось когда-либо препоручать кому-то важную бизнес-функцию, то вы должны представлять себе, каким кошмаром является аутсорсинг. В отсутствие непосредственного контроля за обслуживанием клиентов ваша служба работы с клиентами будет ужасной – из тех, которые описываются в веблогах, когда люди рассказывают, как они пытались добраться *хоть до кого-то* в телефонной компании, чтобы сделать какие-то элементарные вещи. Если вы передаете кому-то реализацию и у вашего партнера иное, чем у вас, представление о том, что такое быстрая доставка, то вашим покупателям будет не очень уютно, и вы ничего с этим не поделаете, потому что, во-первых, у вас ушло три месяца на поиск партнера по реализации, а во-вторых, вы даже не будете *знать*, что ваши клиенты недовольны, потому что они не могут с вами общаться, потому что вы передали подрядчику центр обслуживания клиентов именно для того, чтобы *не* общаться со своими клиентами. Движок электронной коммерции, купленный вами? Он никогда не станет таким гибким, как у Amazon с его *obidos*, который они написали сами. (А если станет, то у Amazon не окажется преимуществ перед конкурентами, которые купили то же самое.) И никакой готовый веб-сервер не будет работать с такой ослепительной скоростью, как тот, что Google сделал своими руками и сам оптимизировал.

К несчастью, этот принцип оказывается в прямом противоречии с идеальным «повторно использовать код хорошо, а изобретать колесо плохо».

Я дам такой совет:

**Если какая-то функция является главной для вашего бизнеса, сделайте ее сами.**

Определите, какая область и какие задачи главные для вашего бизнеса, и сделайте их своими силами. Если вы производите программное обеспе-

чение, то ваш успех связан с написанием отличного кода. Передайте подрядчикам столовую компании и тиражирование CD-ROM. Если у вас фармацевтическая компания, напишите программы для исследования медикаментов, но не пишите собственный бухгалтерский пакет. Если у вас платная веб-служба, напишите собственный бухгалтерский пакет, но не создавайте сами рекламные объявления. Если у вас есть клиенты, никому не передавайте их обслуживание.

Если вы разрабатываете компьютерную игру и вашим преимуществом в конкуренции является сюжет, можно использовать чужую библиотеку 3D-графики. Но если вы собираетесь выделиться необычными 3D-эффектами, то лучше написать свою библиотеку.

Исключение, как мне кажется, есть только одно: ваши сотрудники менее компетентны, чем все остальные, и все, что бы вы ни сделали сами, оказывается халтурой. Таких организаций довольно много. Если вы оказались в одной из них, я вам ничем не могу помочь.

ГЛАВА ТРИДЦАТЬ ШЕСТАЯ

# Первое письмо о стратегии: Ben & Jerry's против Amazon

12 мая 2000 года, пятница

Создаете компанию? Тогда вы должны принять одно очень важное решение, поскольку от него зависит все дальнейшее: что бы вы еще потом ни делали, вам абсолютно *необходимо* определиться, к какому лагерю вы принадлежите, и в соответствии с этим действовать далее, либо вам грозит катастрофа.

Что это за решение? Решите, каким будет ваше развитие – медленным, органичным, прибыльным, или оно будет похоже на взрыв – с очень быстрым ростом и массой капитала.

Естественная модель – начать с небольших ограниченных задач и постепенно, в течение длительного времени, строить свой бизнес. Назову это моделью Ben & Jerry's, потому что эта компания хорошо в нее вписывается.

Другая модель, популярно называемая большим скачком, или захватом территории, требует собрать значительный капитал и как можно быстрее вырасти, не заботясь о прибыльности. Я назову это моделью Amazon, потому что Джефф Безос, основатель Amazon, особенно прославился как ее поборник.

Посмотрим на некоторые различия между этими моделями. В первую очередь надо поинтересоваться: существует ли конкуренция в том бизнесе, которым вы хотите заняться?

| Ben & Jerry's | Amazon |
| --- | --- |
| Множество признанных конкурентов | Новая технология, вначале никакой конкуренции |

Если нет никакой реальной конкуренции, как у Amazon, то имеются шансы осуществить успешный захват территории, т. е. как можно быстрее заполучить как можно больше клиентов, чтобы будущим конкурентам было сложно выйти на этот рынок. Но если вы собираетесь заняться производством, где существует несколько конкурентов с прочным положением, то «захват территории» теряет смысл, потому что свою клиентуру вам придется создавать, каким-то образом переманивая клиентуру конкурентов.

Обычно венчурные капиталисты не испытывают энтузиазма по поводу выхода на рынок, где есть противные конкуренты. Лично меня не так пугает существующая конкуренция – возможно потому, что я работал над Microsoft Excel как раз в тот период времени, когда ей удалось почти целиком вытеснить с рынка Lotus 123, до того практически господствовавшей на нем. Текстовый процессор номер один Word вытеснил WordPerfect, который ранее вытеснил WordStar, и все они в тот или иной момент оказывались практическими монополистами. И Ben & Jerry's выросли в замечательный бизнес, хотя нельзя сказать, что до их появления негде было купить мороженое. Вытеснить конкурента не так уж невозможно, если поставить себе такую цель. (О том, как это сделать, я расскажу в следующих главах.)

Другая проблема вытеснения конкурентов связана с эффектом сети и закреплением клиентов (lock-in):

| Ben & Jerry's | Amazon |
|---|---|
| Отсутствие сетевого эффекта, слабое закрепление клиентов | Сильный сетевой эффект, сильное закрепление клиентов |

*Сетевой эффект* – это такое положение, когда чем больше у вас клиентов, тем больше их приходит. Он основан на законе Меткалфа:[1] стоимость сети пропорциональна квадрату числа пользователей.

Хорошим примером служит eBay. Если вы хотите продать старые часы Patek Philippe, то самую высокую цену вы найдете на eBay, потому что там больше покупателей. Если вы хотите купить часы Patek Philippe, поищите на eBay, потому что там больше продавцов.

Очень сильный сетевой эффект создают фирменные чат-системы типа ICQ или AOL Instant Messenger. Для того чтобы пообщаться в чате, надо идти туда, где есть пользователи чатов, а у ICQ и AOL их больше, чем где бы то ни было. Скорее всего, ваши друзья предпочли одну из этих служб, а не бо-

[1] См. *www.mgt.smsu.edu/mgt487/mgtissue/newstrat/metcalfe.htm*.

лее мелкую, например MSN Instant Messenger. При всех своих деньгах, мощи и маркетинговом опыте Microsoft не в состоянии прорваться на рынок аукционов или чатов из-за сильного сетевого эффекта.

*Закрепление клиентов* – это такая особенность бизнеса, когда люди не склонны менять продавца услуги. Например, никто не хочет менять интернет-провайдера, даже если качество обслуживания не очень нравится, из-за неудобства, связанного с необходимостью оповестить всех о новом почтовом адресе. Люди не хотят переходить на новый текстовый процессор, если он не может читать их прежние файлы.

Еще лучше, чем закрепление, действует такая хитрая его разновидность, которую я называю *скрытым закреплением*: вы даже не замечаете, что привыкли к некой службе. Примером служат все эти новые службы типа PayMyBills.com, которые получают вместо вас все счета, сканируют их и дают возможность просматривать в Интернете. Обычно они обслуживают бесплатно первые три месяца. Но когда эти три месяца подходят к концу и вы не хотите продлевать обслуживание, вам остается только связаться с каждым, кто выставляет вам счета, и попросить его снова высылать счета на ваш домашний адрес. Необходимость сделать это может предотвратить отказ от услуг PayMyBills.com – пусть уж лучше они забирают с моего банковского счета $8,95 ежемесячно. Молодцы!

Если вы входите в бизнес, которому естественно присущи сетевые эффекты и закрепление и в котором нет установившихся конкурентов, то *лучше* выбрать модель Amazon, иначе это сделает кто-то другой, и для вас места не останется.

Берем конкретный пример. В 1998 году AOL тратила громадные средства, благодаря чему каждые пять недель на 1 миллион увеличивала количество своих клиентов. У AOL есть такие замечательные вещи, как комнаты для чатов и instant messaging, обеспечивающие скрытую привязку клиентов. Если вы нашли группу друзей для чата, то *не станете* менять интернет-провайдера. Это все равно, что сменить всех своих друзей. По-моему, это главная причина, по которой AOL может назначать цену 22 доллара в месяц, когда есть масса провайдеров, берущих 10 долларов.

Когда я работал в Juno, тамошнее руководство было просто не в состоянии понять такие факторы, и они упустили отличную возможность обогнать AOL во время захвата территории, когда все начали подключаться к Интернету, – они вкладывали недостаточно средств в приобретение клиентов, потому что не хотели уменьшать доли существующих акционеров, привлекая дополнительный капитал, и не думали стратегически о чатах и IM,

поэтому не разработали никаких функций программного обеспечения для скрытой привязки клиентов, как в AOL. В результате у Juno сейчас около 3 миллионов клиентов, которые платят примерно по $5,50 в месяц, против примерно 21 миллиона у AOL, приносящих в среднем по $17 в месяц. Вот так!

| Ben & Jerry's | Amazon |
|---|---|
| Нужен небольшой капитал; окупаемость быстрая | Нужен огромный капитал; получение прибыли откладывается на годы |

Компании типа Ben & Jerry's начинаются со средств на чьей-то кредитной карточке. В первые месяцы и годы они должны применять бизнес-модель, которая очень быстро начинает приносить прибыль, но может не соответствовать той конечной модели, которая их интересует. Например, вам *хочется* стать гигантской компанией продажи мороженого с годовым объемом продаж 200 млн. долларов, но пока вы должны *довольствоваться* открытием маленького магазина в Вермонте и надеждой, что он принесет прибыль, которую можно будет реинвестировать для расширения бизнеса. История фирмы Ben & Jerry's рассказывает, что они начали с капитала в $12 000. ArsDigita говорит, что они начали с $11 000. Типичные цифры лимита на карточке MasterCard.

Компании типа Amazon собирают деньги примерно с такой скоростью, с какой их можно потратить. Это объяснимо. Они страшно спешат. Если это бизнес, в котором нет конкурентов и есть сетевые эффекты, то они должны очень спешить. Важен каждый день. И есть *масса* способов обратить деньги в выигрыш времени. Почти все они интересны.

- Арендовать готовые меблированные офисные помещения вместо обычного пустого пространства для офиса. Затраты: примерно втрое дороже. Экономия времени: от нескольких месяцев до года в зависимости от рынка.
- Платить немыслимые зарплаты или предлагать программистам BMW в качестве подъемных. Затраты: примерно 25% дополнительно на технический персонал. Экономия времени: можете заполнить вакансии в течение трех недель вместо обычных шести месяцев.
- Нанимать консультантов вместо постоянных служащих. Затраты: примерно втрое дороже. Экономия времени: консультанты могут сразу начать работать. (Консультанты уделяют вам меньше времени

и внимания, чем хотелось бы? Платите им наличными, чтобы они хотели работать только на вас.)

- Не стесняйтесь платить наличными, чтобы решать проблемы на месте. Если ваш новый суперпрограммист меньше работает из-за того, что ему нужно найти новый дом и переехать, подрядите хорошую фирму, которая сделает это для него. Если в новых помещениях долго не могут установить телефоны, купите пару десятков сотовых телефонов. Работу замедляет доступ в Интернет? Найдите двух дополнительных провайдеров. Наймите консьержа, который будет относить вещи сотрудников в химчистку, заказывать номера в гостинице, машины до аэропорта и т. д.

Компании типа Ben & Jerry's не могут позволить себе ничего подобного, поэтому они должны приготовиться к медленному развитию.

| Ben & Jerry's | Amazon |
|---|---|
| Важна корпоративная культура | Корпоративная культура невозможна |

Когда ваша компания растет быстрее, чем на 100 процентов в год, просто невозможно, чтобы наставники передавали новичкам корпоративную культуру. Если программист перемещен на должность руководителя и получил вдруг пятерых новых подчиненных, набранных за день до этого, просто невозможно осуществлять такое наставничество. Netscape являет собой один из самых вопиющих примеров в этом смысле, увеличив за год численность прораммистов с 5 до примерно 2000. В результате их культура оказалась смесью различных представлений о компании различных людей, тянущих в разных направлениях.

Для некоторых компаний в этом может не быть ничего страшного. Для других же корпоративная культура составляет важную часть всего их существования. Ben & Jerry's *существует* благодаря ценностям основателей, которые не согласились бы на более быстрое расширение, чем такое, при котором может распространяться культура.

Рассмотрим гипотетический пример с программированием. Допустим, что вы хотите прорваться на рынок текстовых процессоров. По-видимому, Microsoft в значительной мере закрепила этот рынок за собой, но вы обнаруживаете нишу для тех, кто по каким-то причинам абсолютно не приемлет возможность аварии текстового процессора. Вы решили сделать сверхнадежный профессиональный текстовый процессор, который про-

сто не может аварийно завершиться, и продаете его тем, чья *жизнь* зависит от текстовых процессоров. (Да, это преувеличение. Я же сказал, что пример гипотетический.)

Допустим теперь, что ваша корпоративная культура включает в себя все виды технологий, применяемых для создания высоконадежного кода: тестирование отдельных блоков, формальное оценивание кода, соглашения по кодированию, большие отделы контроля качества и т. д. Все эти технологии нетривиальны, и для их освоения требуется некоторое время. Пока новый программист учится, как писать надежный код, его должен учить и направлять кто-нибудь более опытный.

Как только вы попытаетесь расширяться быстрее, чем допустимо для сохранения такого обучения и наставничества, вы перестанете передавать дальше эти ценности. Вновь принятые на работу программисты не будут получать никакой дополнительной подготовки и станут писать ненадежный код. Они не станут проверять значение, возвращаемое `malloc()`, и в их коде станут возникать сбои при каких-то непонятных обстоятельствах, которых они не предвидели, некому будет посмотреть их код и показать, как правильнее было бы его написать, и все ваши преимущества в конкуренции с Microsoft Word окажутся утраченными.

| Ben & Jerry's | Amazon |
|---|---|
| Ошибки становятся ценными уроками | Ошибок просто не замечают |

Слишком быстро растущая компания просто не замечает своих крупных ошибок, особенно относящихся к перерасходу средств. Amazon покупает Junglee, службу сравнительной выгодности покупок, за 180 000 000 в акциях и тут же обнаруживает, что службы сравнительной выгодности покупок не очень хороши для ее бизнеса, а потому просто закрывает ее. Когда денег невпроворот, глупые ошибки легко скрывать.

| Ben & Jerry's | Amazon |
|---|---|
| Превращение в крупную компанию длится долго | Становится крупной компанией очень быстро |

Быстрый рост создает *впечатление* успеха (если только не сам успех). Когда предполагаемые служащие видят, что вы нанимаете по 30 человек в неделю, им начинает казаться, что они – участники чего-то крупного, захватывающего, успешного, что окончится выпуском ценных бумаг. А «ма-

ленькая сонная компания» с 12 работниками и собакой может не произвести на них такого сильного впечатления, даже если это сонное царство приносит прибыль и в будущем разовьется в лучшую компанию.

Практическое правило говорит, что можно создать прекрасные условия для работы, а можно пообещать людям, что они быстро разбогатеют. Но вы должны выбрать что-то одно, иначе никого не наймете на работу.

На некоторых из ваших служащих произведет впечатление, что компания имеет высокие шансы выпустить на рынок свои ценные бумаги и раздает щедрые опционы на акции. Эти люди будут готовы вложить в компанию три или четыре года своего труда, хотя и ненавидят каждую минуту, отданную работе, – потому что надеются сорвать куш.

В случае медленного и естественного развития материальный выигрыш может оказаться на большем отдалении. Остается только создать такую рабочую обстановку, в которой само путешествие оказывается наградой. Здесь не может быть лихорадочных 80-часовых рабочих недель. Офис не может быть большим шумным верхним этажом, битком набитым складными столами и жесткими деревянными стульями. Людям должны предоставляться приличные отпуска. Они должны быть друзьями, а не просто коллегами по работе. Играют роль социология и общность. Менеджеры должны быть цивилизованными и не стоять над душой у людей, как микроменеджеры у Дилберта. Если вы все это сделаете, то привлечете множество людей, которых не раз дурачили обещаниями сделать их миллионерами при начальном размещении акций на бирже; теперь они ищут что-нибудь *надежное*.

| Ben & Jerry's | Amazon |
|---|---|
| Возможно, вы добьетесь успеха. Во всяком случае вы не потеряете много денег | У вас есть крохотный шанс стать миллионером и большие шансы провалиться |

Модель Ben & Jerry's, если вести себя достаточно умно, должна привести к успеху. Возможно, придется побороться, одни годы будут удачнее, чем другие, но если не наступит еще одна Великая Депрессия, то вы наверняка не потеряете *слишком* много денег, в первую очередь потому, что и вложили не много.

Беда с моделью Amazon в том, что все только и думают, что об Amazon. А Amazon существует только один. Думать надо об остальных 95% компаний, которые растрачивают безумное количество венчурного капитала,

а потом просто проваливаются, потому что их продукты никому не нужны. Во всяком случае, если вы последуете модели Ben & Jerry's, то выясните, что ваш продукт никому не нужен задолго до того, как истратите больше лимита кредита по одной MasterCard.

## *Что хуже всего?*

Хуже всего, если вы так и не решите, какая у вас будет компания: Ben & Jerry's или Amazon.

Если вы вступаете на рынок, где нет конкуренции, закрепления клиентов и эффектов сети, *лучше принять модель* Amazon, или вас ждет судьба Wordsworth.com, который начал за два года до Amazon и о котором никто не слышал. Или еще хуже, вы станете сайтом-призраком вроде MSN Auctions,[1] практически не имеющим шансов побороть eBay.

Если вы вступаете на уже устоявшийся рынок, то выбор стратегии быстрого роста станет увлекательным способом спустить тонны денег, как это сделал BarnesandNoble.com. Лучшее, на что вы можете надеяться, – это создать что-то *надежное* и *прибыльное* и годами медленно продвигаться в конкурентной борьбе.

Никак не решить? Надо учитывать и другие факторы. Подумайте о своих личных ценностях. Какую компанию вам больше хотелось бы иметь – похожую на Amazon или похожую на Ben & Jerry's? Для начала прочтите пару историй компаний Amazon[2] и Ben & Jerry's,[3] несмотря на то, что это жития святых в чистом виде, и посмотрите, какая из них больше соответствует вашему набору жизненных ценностей. На самом деле еще лучшей моделью для компании типа Ben & Jerry's является Microsoft, и историй Microsoft написано множество. Microsoft в некотором смысле «повезло» со сделкой по PC-DOS, но компания была прибыльна и постоянно расширялась, поэтому она могла очень долго дожидаться своего большого прорыва.

---

1 См. *auctions.msn.com/*.

2 Robert Spector (Роберт Спектор) «amazon.com – Get Big Fast: Inside the Revolutionary Business Model that Changed the World» (amazon.com--быстрый рост: взгляд изнутри на революционную бизнес-модель, изменившую мир), HarperCollins, 2000.

3 Fred Lager (Фред Лагер) «Ben & Jerry's: The Inside Scoop: How Two Real Guys Built a Business With a Social Conscience and a Sense of Humor» (Ben & Jerry's: как два отличных парня создали бизнес с социальной ответственностью и чувством юмора), Crown, 1994.

Подумайте о своем отношении к риску и успеху. Вы хотите попытаться стать миллиардером к 35 годам, даже если по сравнению с вероятностью успеха этой затеи игра в лотерею выглядит как надежная сделка? Компании типа Ben & Jerry's такой возможности вам не предоставят.

Наверное, хуже всего, если вы решите стать компанией типа Amazon, а действовать станете, как компания типа Ben & Jerry's (никогда в этом себе не признаваясь). Компаниям типа Amazon абсолютно *необходимо* при каждом случае жертвовать деньгами, чтобы выиграть время. Вам может показаться, что вы разумны и бережливы, когда настаиваете на поиске программистов, которые станут работать по среднерыночным тарифам. Нет, вы не так разумны, потому что у вас уйдет на это шесть месяцев вместо двух, а эти четыре месяца могут означать, что вы пропустили сезон рождественских покупок, а значит, это обошлось вам в целый год, а возможно, и весь ваш бизнес-план стал неосуществим. Вы можете счесть разумным сделать версию вашего продукта для Макинтошей вместе с Windows-версией, но что если подготовка к выпуску продукта займет вдвое дольше времени, поскольку программистам надо создать слой совместимости, а покупателей в результате вы привлечете всего на 15% больше? Покажется ли вам ваше решение таким уж разумным?

Обе модели эффективны, но выбрать надо одну из них и придерживаться ее, иначе вы столкнетесь с тем, что по каким-то таинственным и непонятным вам причинам дела пойдут плохо.

ГЛАВА ТРИДЦАТЬ СЕДЬМАЯ

# Второе письмо о стратегии: что сначала – курица или яйцо

24 мая 2000 года, среда

Идея рекламы состоит в том, чтобы врать и не быть пойманным. Большинство фирм, начиная рекламную кампанию, просто берет самую неприятную правду, которая их характеризует, переворачивает ее наизнанку («ложь») и отправляет эту ложь в цель. Назовем это «доказательством с помощью многократного повторения». Допустим, например, что в самолете тесно и неудобно, а служащие авиакомпании грубы и нелюбезны, да и вся *система* коммерческих авиалиний служит источником мучений. Поэтому большая часть рекламы авиалиний старается убедить, насколько *удобно* и *приятно* летать и как о вас будут *заботиться* на каждом шагу. Когда British Airways показала рекламу, в которой бизнесмен в самолете видел себя во сне ребенком, спящим в корзине, это превзошло все разумные границы.

Еще примеры? Производящие бумагу компании вырубают древние деревья, опустошая леса, на которые у них даже нет права собственности. Соответственно, в рекламе они обязательно покажут какой-нибудь прекрасный старый сосновый лес и поговорят о том, как сильно они заботятся об окружающей среде. Сигареты являются причиной смерти, поэтому на их рекламе изображаются типичные счастливые улыбающиеся здоровые люди, занимающиеся спортом на свежем воздухе. И так далее.

Когда появился первый Макинтош, для него не было никакого программного обеспечения. Неудивительно, что Apple сделала огромный блестящий каталог с перечислением всего потрясающего программного обеспечения, которое было «доступно». Половина перечисленных продуктов была помечена мелким шрифтом как находящаяся «в процессе разработки», а оставшуюся половину невозможно было достать никакими сред-

ствами. Некоторые продукты были настолько кривыми, что их никто бы и не купил. Но даже наличие толстого блестящего каталога с программным «продуктом» на каждой странице, описанным пылким слогом, не могло скрыть того, что невозможно было купить текстовый процессор или электронную таблицу для работы на Макинтоше со 128 килобайтами памяти. Аналогичные «руководства по программному обеспечению» существовали для NeXT и BeOS. (Вниманию поклонников NeXT и BeOS: не будем пререкаться по поводу ваших восхитительных операционных систем? Пишите в своих собственных колонках.) Единственное, о чем вас информирует руководство по программному обеспечению, так это об отсутствии программного обеспечения для данной системы. Увидев его, бегите без оглядки.

Amiga, Atari ST, Gem, IBM TopView, NeXT, BeOS, Windows CE, General Magic – перечню провалившихся «новых платформ» нет конца. Поскольку это *платформы*, то по определению они не так интересны сами по себе – без программного обеспечения, которое выполнялось бы на них. Однако, за очень немногими исключениями (и я уверен, что получу целую *кучу* почты от нудных сторонников загадочных и нелюбимых платформ типа Amiga или RSTS-11), никакой разработчик программ, у которого есть хоть капля здравого смысла, не станет сознательно писать программы для платформ, у которых в *лучшие* времена было по 100 000 пользователей, такой как BeOS, когда с такими же затратами труда можно написать программу для платформы с 100 000 000 пользователей, такой как Windows. Тот факт, что вообще находятся те, кто пишет для этих оригинальных платформ, доказывает, что не все в мире движется жаждой наживы: религиозный пыл никуда не исчез. Ради бога. Ты написал чудесный клон микро-Emacs для Timex Sinclair 1000. Браво. Вот тебе четвертак, пойди купи себе что-нибудь.

Если вы занялись созданием платформ, то, вероятно, столкнетесь с тем, что обычно называют *парадоксом курицы и яйца*: никто не станет покупать платформу, пока для нее не будет хорошего программного обеспечения, и никто не станет писать программное обеспечение, пока у платформы не будет достаточно широкой установочной базы.[1] Вот так. Что-то вроде гордиева узла, хотя лучше было бы назвать это «гордиевой спиралью смерти».

Парадокс курицы и яйца и его варианты – это самый важный элемент стратегии, который необходимо понять. Возможно, вы проживете и не по-

[1] Это хорошо известно, но только никто даже ориентировочно не берется количественно оценить «достаточно широкую установочную базу» – сколько это: 100 000, 1 000 000, 10 000 000 или больше? – *Примеч. науч. ред.*

нимая его: Стив Джобс практически сделал *карьеру* на том, что не понял проблему курицы и яйца, причем *дважды*. Но у нас нет в распоряжении такого Персонального Поля Искажения Реальности, как у Джобса, поэтому нам придется, сгорбившись, напряженно учиться.

Классическое царство парадокса курицы и яйца – это программные платформы. Но есть еще одна область. Каждый месяц *миллионы* компаний кредитных карт отправляют по почте клиентам *мириады* счетов. Клиенты выписывают на бумаге чеки, вкладывают их в *триллионы* конвертов и отправляют по почте обратно. Эти конверты помещают в большие коробки и отправляют в страны с дешевой рабочей силой, где их откроют и обработают. Однако вся операция не дешева: по последним попавшимся мне данным – более 1 доллара за счет.

Нам, привыкшим к Интернету, это кажется смешным. «Пришлите мне мой счет электронной почтой, – говорите вы, – я оплачу его в онлайне». Вы говорите: «Это будет стоить какую-нибудь 1/100000 часть пенни. Вы сэкономите *миллионы*». Или что-нибудь в этом роде.

И вы правы. Поэтому многие компании попытались проникнуть в эту область, которая технически называется *предъявлением счетов*. Одна из них (угадайте, кто?) Microsoft. Их решение, TransPoint,[1] выглядит так: это веб-сайт, на который вы заходите, видите там свои счета и оплачиваете их.

Итак, если вы станете получать счета через эту систему Microsoft, вам придется каждые несколько дней заходить на сайт и проверять, нет ли новых счетов, чтобы не пропустить их. Если вы получаете всего десяток счетов в месяц, это может быть не слишком обременительно. Но здесь кроется другая неприятность: лишь немногие компании станут выставлять вам счета через эту систему. Другие счета вам придется искать где-то в другом месте.

Конечный результат? Оно того не стоит. Меня удивит, если системой пользуется хотя бы 10000 человек. Дальше Microsoft обращается в компании и говорит им: «Посылайте своим клиентам счета через нашу систему!» А компании отвечают: «OK! Сколько это будет стоить?» А Microsoft отвечает: «50 центов! И это гораздо меньше, чем 1 доллар!» А компании говорят: «OK. И это все?» А Microsoft говорит: «Ах, да! Вам надо еще потратить $250 000 на то, чтобы установить программное обеспечение, соединить ваши системы с нашими и запустить все это в работу».

А так как у Microsoft ничтожно мало пользователей этой системы, трудно представить себе, что кто-то станет платить $250 000, чтобы сэконо-

[1] TransPoint давно умерла. Похоже, я был прав.

мить по 50 центов на 37 пользователях. Вот! Проблема курицы и яйца подняла свою уродливую голову! Клиенты не придут, пока у вас не будет компаний, а компании не придут, пока у вас не будет клиентов! Microsoft может выйти из этого затруднения при помощи денег.[1] Но для небольших фирм это не выход. Так что же делать?

На самом деле программные платформы дают хорошие советы по разрешению парадокса курицы и яйца. Взглянем-ка на историю платформ персональных компьютеров за годы после появления IBM-PC – возможно, мы там что-нибудь обнаружим!

Большинство людей считает, что для IBM-PC требовалась PC-DOS. Это не так. Когда впервые появился IBM-PC, можно было выбирать из трех операционных систем: PC-DOS, XENIX (слабенькая 8-разрядная версия UNIX, выпущенная – я не обманываю – Microsoft) и нечто под названием UCSD P-System,[2] несколько напоминавшее, если вы готовы в это поверить, Java: замечательные, медленные, переносимые байт-коды за 20 лет до Java.

Сегодня большинство людей никогда не слышало о XENIX или об оригинальной разработке UCSD. Наверное, многие думают, что во всем виновата Microsoft, захватившая с помощью своей маркетинговой мощи или чего-то еще рынок маленьких операционных систем. А вот и нет: Microsoft в те дни была крохотной компанией. У кого была маркетинговая мощь, так это у Digital Research, которая производила другую операционную систему. Так почему же PC-DOS оказалась победителем в гонке?

До появления PC единственной операционной системой, которая существовала реально, была CP/M, но рынок компьютеров с CP/M, стоивших примерно $10 000, был очень мал. Капризные и дорогие, они были не слишком дружественны к пользователю. Но те, кто их покупал, делали это для работы с текстовым процессором, потому что можно было купить неплохой текстовый процессор WordStar для CP/M, а Apple II *не мог* обрабатывать текст (начать с того, что там не было букв нижнего регистра).

Вот малоизвестный факт: даже DOS 1.0 была спроектирована со *встроенным* режимом обратной совместимости с CP/M.[3] У нее не только был

---

1 Похоже, я был неправ.

2 См. *www.threedee.com/jcm/psystem/*.

3 Более того, справедливо будет сказать, что MS DOS 1.0 была спроектирована как прямое заимствование из CP/M многого и многого, например формата исполняемых COM-файлов. DOS должна была вытеснить существенно более развитую к тому времени CP/M и сделать это в экстремально короткий срок, иначе у нее не было шансов на выживание... – *Примеч. науч. ред.*

собственный элегантный интерфейс программирования, известный закоренелым программистам как INT 21, но она полностью поддерживала прежний программный интерфейс CP/M. Она *почти* могла выполнять программы для CP/M. На самом деле WordStar был перенесен под DOS путем изменения *одного-единственного байта* кода (Настоящий Программист скажет вам, что это был за байт, а я уже давно забыл). Не будет лишним повторить: WordStar был перенесен под DOS путем изменения *одного-единственного байта* кода. Это надо хорошенько осознать.

Так.

Понятна мысль?

DOS стала популярной, потому что *с первого же дня для нее были программы*. А программы были потому, что Тим Патерсон подумал о включении в нее совместимости с CP/M, поскольку в те давние времена были умные люди, понимавшие парадокс курицы и яйца.

Быстро перенесемся вперед. Вся *история* платформы PC знала лишь два крупных изменения парадигмы, затронувших почти каждого пользователя PC: мы все перешли на Windows 3.x, а затем мы все перешли на Windows 95. Лишь крайне малое количество пользователей переходило за это время на что-либо еще. Тайные замыслы Microsoft для захвата мира? Если хотите, можете так считать. Я думаю, что причина была другой, более интересной, возвращающей нас к курице и яйцу.

*Мы все перешли на Windows 3.x*. Важно обратить в этом предложении внимание на цифру 3. Почему мы все не перешли на Windows 1.0? Или Windows 2.0? Или Windows 286 и затем Windows 386? Потому что Microsoft нужно сделать пять версий, чтобы «довести до ума» систему? *Нет*.

Настоящая причина еще глубже и связана с загадочными аппаратными возможностями, впервые появившимися в процессоре Intel 80386, который был необходим для Windows 3.0.

- Первая возможность. Старые программы DOS, чтобы вывести что-нибудь на экран, выполняли запись прямо по адресам памяти, соответствующим позициям символов на экране. Только так можно было выполнять вывод достаточно быстро, чтобы программа выглядела прилично. Но Windows выполнялась в графическом режиме. На прежних процессорах Intel разработчикам из Microsoft приходилось переключаться в полноэкранный режим, когда нужно было выполнить DOS-программу. Но на 80386 можно было создать блоки виртуальной памяти и установить прерывания, с тем чтобы опера-

ционная система получала *уведомление* всякий раз, когда программа пыталась что-то писать в память экрана. Windows могла в этом случае мгновенно записать эквивалентный текст в графическое окно на экране.

- Вторая возможность. Старые программы DOS считали, что полностью владеют процессором. В результате они плохо уживались вместе. Но Intel 80386 позволял создавать «виртуальные» PC, каждый из которых действовал, как отдельный 8086, поэтому старые программы для PC могли считать, что полностью распоряжаются компьютером, несмотря на то, что одновременно выполнялись другие программы, которые тоже считали, что они владеют всем компьютером.

Поэтому Windows 3.x на Intel 80386 была первой версией, которая могла прилично выполнять несколько программ DOS. (Технически Windows 386 тоже могла, но 80386 были редки и дороги примерно до той поры, когда вышла Windows 3.0.) Windows 3.0 была первой версией, которая действительно могла делать важную работу – выполнять все старые программы.

Windows 95? Без проблем. Прекрасный новый 32-разрядный API, но старые 16-разрядные приложения отлично выполняются. Microsoft была этим очень озабочена и потратила много времени, тестируя Windows 95 со всеми старыми программами, которые только можно было найти. Джон Росс, написавший первоначальную версию SimCity для Windows 3.x, рассказывал мне, что у него была случайная ошибка в SimCity, когда он читал память, которую только что освободил. Под Windows 3.x все работало прекрасно, потому что память никуда не уходила. Но вот что поразительно. При тестировании бета-версий Windows 95 обнаружилось, что SimCity не работает. В Microsoft нашли ошибку и *добавили в Windows 95 специальный код, который проверял, не выполняется ли SimCity*. При обнаружении SimCity выделение памяти переводилось в особый режим, в котором память не освобождалась сразу. Эта помешанность на обратной совместимости способствовала переходу пользователей на Windows 95.[1]

Теперь у вас должно появиться некоторое представление о том, как решается парадокс курицы и яйца: обеспечьте режим обратной совместимо-

[1] Эта же приверженность обратной совместимости, из-за которой фрагменты 16-разрядного кода еще долгие годы сохранялись в декларируемой 32-разрядной операционной системе, намного замедлила прогресс Win32 API, который сформировался «в недрах» системы Windows NT, менее обремененной условиями совместимости. – *Примеч. науч. ред.*

сти, осуществляющий доставку грузовика куриц или грузовика яиц, смотря как поглядеть на это, после чего можете расслабиться и купаться в деньгах.

Ну ладно, вернемся к предъявлению счетов. Вспомнили? Парадокс курицы и яйца в том, что если вы сможете получать счета только от одной компании, то не станете пользоваться этой службой. Как преодолеть затруднение? В Microsoft не догадались. PayMyBills.com (и еще полдюжины новых компаний в Силикон Вэлли) догадались все одновременно. Вы создаете *режим обратной совместимости*: если компания не хочет участвовать в электронной системе, так пусть она посылает свои чертовы счета на Университетскую авеню в Пало-Альто, где сидят живые люди, которые откроют конверты и введут счета через сканер. Теперь уже вы увидите *все* свои счета на их веб-сайте. Поскольку через эту систему доступна любая компания, которая только существует, клиенты удовлетворены, даже если система работает в этом дурном режиме, когда глупые банки-члены Visa посылают электронный счет на принтер, печатают его на бумаге, вкладывают в конверт, посылают за 1500 миль в Калифорнию, где конверт вскрывают, выбрасывают на помойку дурацкие рекламные листки, предлагающие никчемные «бесплатные» часы-приемник, в действительности стоящие $9,95, а бумажный счет вводят через сканер в компьютер и выкладывают в Интернет, куда его нужно было отправить сразу. Но дурацкий режим обратной совместимости в конечном итоге отомрет, потому что PayMyBills.com, в отличие от Microsoft, может действительно привлечь клиентов к использованию своей системы, поэтому вскоре они смогут пойти к глупым банкам-членам Visa и сказать: «Послушайте, у меня 93 400 ваших клиентов. Почему бы вам не экономить $93 400 каждый месяц, подключившись ко мне напрямую?» И PayMyBills.com станет внезапно очень прибыльной, тогда как Microsoft все еще трудится над подключением к своей системе второй энергокомпании – может быть, для разнообразия выбрать ту, которая обслуживает Джорджию?

О компаниях, не видящих парадокса курицы и яйца, я бы сказал, что они собираются *вскипятить море*: в соответствии с их бизнес-планом нужно, чтобы 93 000 000 человек стали участвовать в их сумасшедшей бизнес-схеме, чтобы она заработала. Одна из наиболее глупых идей из всех, когда-либо встречавшихся мне, называлась ActiveNames.[1] Она состояла в

[1] У ActiveNames ничего не вышло. См. статью Патриции Оделл (Patricia Odell) «ActiveNames Shutters Business» (ActiveNames закрывает бизнес), DIRECT Newsline, April 17, 2001 (*www.directmag.com/ar/marketing_activenames_shutters_business/*).

том, что все живущие на свете люди захотят добавить к своей программе почтового клиента маленький модуль, который станет по фамилии искать реальный почтовый адрес на их центральном сервере. То есть вы не будете говорить кому-то, что ваш почтовый адрес *kermit@sesame-street.com*, а скажете ему, что на ActiveName вас зовут «spolsky», и если они хотят написать вам, то они должны установить эту специальную программу. Бр-р-р-р. Я не могу даже начать перечислять те причины, по которым эта идея никогда не будет работать.

Вывод: Если на вашем рынке существует парадокс курицы и яйца, то *надо* обеспечить режим обратной совместимости, который разрешит этот парадокс, иначе вам придется о-о-очень долго ждать, пока ваша система заработает (скорее всего, вечно).

Есть много других компаний, понявших, что столкнулись с парадоксом курицы и яйца, и разрешивших его разумным образом. Когда Transmeta представила свой CPU, это был первый за *долгое* время случай, когда некая компания, *не* Intel, признала, наконец, что если у вас есть CPU и вы хотите, чтобы его покупали миллионами, он должен выполнять код x86. Это случилось после того, как Hitachi, Motorola, IBM, MIPS, National Semiconductor и Бог знает сколько других компаний соблазнились мыслью, что у них есть право изобрести новый набор команд. Архитектура Transmeta с самого начала признает, что любой бизнес-план, призывающий к созданию компьютера, на котором не может выполняться Excel, ведет в никуда.

ГЛАВА ТРИДЦАТЬ ВОСЬМАЯ

# Третье письмо о стратегии: пустите меня обратно!

3 ИЮНЯ 2000 ГОДА, СУББОТА

Если вы хотите, чтобы люди бросили конкурирующий продукт и перешли на ваш, то должны понимать, что *препятствует вашему выходу* на рынок, и понимать очень хорошо, потому что никто никуда не перейдет, а вы останетесь не у дел.

Пару глав назад я писал о разнице между компаниями двух типов: типа Ben & Jerry's, пытающейся продвинуться на рынке со сложившейся конкуренцией, и типа Amazon.com, пытающейся «захватить территории» в новой области, где конкуренция не сложилась. Когда я работал над Excel в начале 1990-х, Microsoft была типичным представителем лагеря Ben & Jerry's. У Lotus 123, признанного конкурента, была почти полная монополия на рынке электронных таблиц. Конечно, некоторые новые пользователи, купив компьютер, сразу начинали с Excel, но если Microsoft действительно хотела продавать электронные таблицы, то должна была сделать переход на свой продукт привлекательным для пользователей.

Самое главное в такой ситуации – *признать этот факт*. Некоторые компании не могут сделать даже этого. Руководство предыдущей фирмы, где я работал, Juno, не желало признать, что AOL уже достигла доминирующего положения. Они говорили, что «миллионы людей еще не подключились к Интернету». Они говорили, что «на каждом рынке есть место для двух игроков: Time и Newsweek, Coke и Pepsi и т. д.» Единственное, чего они не говорили, это что «мы должны сделать так, чтобы люди хотели перейти к нам из AOL». Мне не совсем понятно, чего они боялись. Возможно, они боялись «разбудить спящего медведя». Один из лучших программистов

Juno (нет, не я) имел *дерзость*, явную *наглость*, задать на совещании простой вопрос: «Почему мы не стараемся переманить к себе клиентов AOL?» Его вызвали, час орали на него и отказали в обещанном повышении. (Угадайте, где нашлось место для его таланта?)

Нет ничего страшного в том, что рынок уже поделен между конкурентами. На самом деле, даже если ваш продукт радикально нов, как eBay, у него может быть конкуренция: распродажа на дому! Не переживайте. Если ваш продукт в каком-то отношении действительно лучше, то у вас есть неплохие шансы переманить клиентов. Но для этого требуется стратегическое мышление, а мыслить стратегически – значит, видеть *на один шаг дальше*.

*Единственная стратегия* добиться перехода на ваш продукт – это *устранение препятствий*. Представьте себе, что сейчас 1991 год. Доминирующая электронная таблица со 100-процентной долей рынка – это Lotus 123. Вы руководитель продукта Microsoft Excel. Задайте себе вопрос: какие есть препятствия для перехода? Что удерживает пользователей от того, чтобы завтра же стать покупателями Excel? Вот некоторые существовавшие в то время препятствия для потенциальных пользователей Excel:

| Препятствие | |
|---|---|
| 1. Они должны больше узнать об Excel и узнать, что она лучше 123. | |
| 2. Они должны купить Excel. | |
| 3. Они должны купить Windows, чтобы работать в Excel. | |
| 4. Они должны преобразовать свои таблицы из 123 в Excel. | |
| 5. Они должны переделать свои макросы, которые не смогут выполняться в Excel. | |
| 6. Они должны изучить новый интерфейс. | |
| 7. Им нужен более мощный компьютер. | |

И так далее и так далее. Можете считать эти помехи полосой препятствий, преодолев которую, пользователи станут вашими клиентами. Если на старт выйдет 1000 бегунов, то примерно половина из них будут ползти, половина оставшихся не сможет перепрыгнуть через стену, следующая половина из оставшихся свалится в грязь с веревочной лестницы и так, пока

не останутся один-два человека, преодолевших все препятствия. Если препятствий восемь или девять, то на том или ином этапе из соревнований выпадут *все*.

Этот расчет показывает, что *устранение препятствий к переходу* – важнейшее условие захвата поделенного рынка, потому что устранение *всего одного препятствия может удвоить* объем продаж. Устраните два препятствия, и вы еще раз удвоите объем продаж. Microsoft рассмотрела список препятствий и взялась за устранение *каждого из них*:

| Препятствие | Способ преодоления |
|---|---|
| 1. Они должны больше узнать об Excel и узнать, что она лучше 123. | Рекламировать Excel, распространять демонстрационные дискеты, устраивать агитационные поездки по стране. |
| 2. Они должны купить Excel. | Предоставить специальную скидку пользователям 123. |
| 3. Они должны купить Windows, чтобы работать в Excel. | Бесплатно поставлять с Excel облегченную версию Windows. |
| 4. Они должны преобразовать свои таблицы из 123 в Excel. | Добавить в Excel возможность чтения таблиц 123. |
| 5. Они должны переделать свои макросы, которые не смогут выполняться в Excel. | Сделать возможным выполнение макросов 123. |
| 6. Они должны изучить новый интерфейс. | Дать возможность работать с клавиатурой в соответствии со схемой, принятой в 123. |

И неплохо получилось. Неустанно работая над преодолением барьеров, они смогли понемногу отбирать рынок у Lotus.

При смене монополий часто наблюдается переход через магическую «поворотную точку»: в одно прекрасное утро вы обнаруживаете, что у вас не 20, а 80% рынка. Такой переворот происходит *очень* быстро (от VisiCalc к 123 к Excel, от WordStar к WordPerfect к Word, от Mosaic к Netscape к Internet Explorer, от dBase к Access и т. д.). Обычно это происходит, когда падает последний барьер и логичным становится всеобщий переход.

Ясно, что необходимо работать над устранением очевидных препятствий для перехода, но, когда вы считаете, что справились с ними, необходимо выяснить, нет ли других, менее очевидных препятствий. И тут стратегия должна стать более изобретательной, потому что некоторые препятствия отнюдь не бросаются в глаза.

Приведу пример. Этим летом я живу большую часть времени в доме на побережье, но мои счета продолжают поступать в квартиру в Нью-Йорке. Кроме того, я много путешествую. Есть замечательная веб-служба PayMyBills.com, призванная упрощать людям жизнь: вы договариваетесь, что *все* ваши счета будут посылаться им, а они станут сканировать счета и выкладывать в Интернет, чтобы вы могли просмотреть их, находясь в любом месте.

Стоимость услуги PayMyBills составляет около $9 в месяц, что вполне приемлемо, и я не прочь воспользоваться ею, но в прошлом у меня был неудачный опыт финансового обслуживания через Интернет, например через Datek, которые делали столько *арифметических* ошибок в моих счетах, что я засомневался, есть ли у них лицензия. Поэтому я хочу на некоторое время *переключиться* на PayMyBills и, если это мне не понравится, вернуться к прежнему положению вещей.

Беда в том, что, попользовавшись PayMyBills и решив, что она мне не подходит, я буду вынужден обзвонить все эти компании кредитных карт и *снова* изменить свой адрес. Это утомительно. И поэтому *страх* перед трудностью *обратного перехода* мешает мне воспользоваться данной службой. Выше я назвал это *скрытым закреплением*[1] и даже похвалил такую находчивость, но если ваши потенциальные клиенты обнаружат его, то вам не повезло.

Это препятствие для перехода. Оно не в трудности перехода, а в трудности *возвращения*.

И это напомнило мне о поворотном моменте для Excel, случившемся где-то в районе версии 4.0. И связан он был главным образом с тем, что Excel 4.0 стала первой версией Excel, которая могла также и *записывать* таблицы Lotus.

Да, вы не ослышались. *Записывать*. Не читать. Оказывается, от перехода на Excel людей удерживало то, что все, с кем они работали, по-прежнему пользовались Lotus 123. Им не нужен был продукт, создававший таблицы, которые никто больше не мог читать: классический парадокс курицы и яйца.[2] Если вы единственный поклонник Excel в компании, где все остальные пользуются 123, то, даже если вы обожаете Excel, вы не можете перейти на нее, пока не вживетесь в среду 123.

---

[1] См. главу 36.

[2] См. главу 37.

Чтобы захватить рынок, надо справиться со *всеми* препятствиями для перехода. Если упустить хотя бы одно, на котором спотыкаются 50% потенциальных клиентов, то вы *по определению* не захватите больше 50% рынка, никогда не вытесните главного игрока и увидите темную сторону парадокса курицы и яйца (омлет).

Трудность в том, что большинство администраторов видит в стратегии в каждый момент лишь один очередной шаг, как шахматисты, отказывающиеся думать более чем на один ход вперед. Большинство из них скажет: «Надо дать возможность конвертировать *в* мой продукт, но зачем тратить мой ограниченный бюджет, чтобы обеспечить конвертирование *из него*?»

Это детский подход к стратегии. Он напоминает независимых книготорговцев, которые говорят: «С какой стати я должен создавать людям удобства для чтения книг в моем магазине? Я хочу, чтобы они покупали, а не читали!» И тогда в один прекрасный день Barnes and Nobles ставит *диваны*, устраивает *кафе* в магазинах и практически *упрашивает* читать в их магазинах книги, не покупая их. И все эти покупатели теперь *часами* сидят в магазине, *лапают* книги грязными руками, и вероятность того, что они найдут то, что им захочется купить, прямо пропорциональна продолжительности времени, проведенного в магазине, и даже самый дрянной магазин Barnes and Nobles в Айова-сити гребет деньги сотнями долларов *в минуту*, а независимые книготорговцы покидают бизнес. Вот в чем дело, Shakespeare and Company на северо-западе Манхеттена закрывается не потому, что у Barnes and Nobles ниже цены; он закрывается потому, что в магазинах Barnes and Nobles *больше человеческих существ*.

Зрелый подход к стратегии исключает насилие над потенциальными клиентами. Если кто-то *еще не стал вашим клиентом*, не стоит пытаться закрепить его. Будет у вас 100% рынка, тогда и поговорим о закреплении. До того времени пытаться закрепить клиентов слишком рано, и если клиенты поймают вас за этим делом, то больше вы их *не увидите*. Никто не станет переходить на продукт, который ограничит их свободу в будущем.

Возьмем более свежий пример: интернет-провайдеры (ISP), рынок с высокой конкуренцией. Практически ни один ISP не предлагает возможность перенаправления вашей почты на другой адрес *после того, как вы прекратите пользоваться его услугами*. Это самый никудышный поверхностный подход, и странно, что никто этого не сообразил. Если вы небольшой интернет-провайдер и пытаетесь привлечь к себе абонентов, то учтите, что самым большим препятствием станет их беспокойство о том, как сообщить своим друзьям новый почтовый адрес. Поэтому они даже попробо-

вать вашу службу не захотят. А если все же попробуют, то какое-то время не станут сообщать друзьям свой новый адрес, на случай, если они у вас не задержатся. А это означает, что они не будут получать на новый адрес много почты, а это значит, что они и не станут по-настоящему испытывать вашу службу, чтобы определить, нравится ли она им. Сплошные убытки.

А теперь допустим, что некий смелый провайдер объявит: «Попробуйте работать с нами. Если вам у нас не понравится, мы сохраним ваш адрес активным и будем бесплатно пересылать вашу почту любому другому ISP. Пожизненно. Перескакивайте от одного провайдера к другому, сколько хотите, только сообщите нам, и мы станем вашей постоянной службой переадресации».

Конечно, с бизнес-менеджером случится припадок. Почему мы должны *облегчать* клиентам отказ от нашей службы? Это из-за близорукости. *Это еще не ваши клиенты*. Попробуйте закрепить их *до того*, как они станут вашими клиентами, и вы *закроете* перед ними дверь. Но если вы честно пообещаете, что им легко будет отказаться от сервиса, если он им не понравится, то устраните еще одно препятствие на пути к их переходу. А, как мы выяснили, устранение даже одного препятствия может оказать драматическое воздействие на обращение в другую веру, а со временем, когда вы снесете последний барьер для перехода, народ к вам повалит, и ваша жизнь станет счастливой – на какое-то время. Пока кто-то не проделает то же самое с вами.

ГЛАВА ТРИДЦАТЬ ДЕВЯТАЯ

# Четвертое письмо о стратегии: bloatware и миф 80/20

3 марта 2001 года, пятница

Версия 5.0 ведущей программы Microsoft для работы с электронными таблицами Excel вышла в 1993 году. Она решительно была *громадной*: ей требовалось целых 15 Мбайт на жестком диске. Тогда мы еще помнили свои первые 20-мегабайтные диски PC (примерно 1985 г.), поэтому, конечно, казалось, что 15 Мбайт – это много.

Когда вышла Excel 2000, ей требовалось чудовищно много – 146 Мбайт, почти в десять раз больше! Черт бы побрал этих безграмотных программистов Microsoft, да?

Нет.

Наверняка вы думаете, что я хочу написать одну из тех нудных статей, лежащих на каждом углу в Интернете и плачущих по поводу «bloatware». Вах-вах-вах, какое все раздутое, о горе мне, edlin и vi гораздо лучше, чем Word и Emacs, потому что они такие стройные и т. п.

Ха-ха! Я вас обманул, потому что не собираюсь писать еще одну такую статью, потому что они ошибочные.

По тогдашним ценам на жесткие диски Microsoft Excel 5.0 занимала дискового пространства на 36 долларов.

В 2000 году, по ценам 2000 года, Microsoft Excel 2000 занимает дискового пространства на 1,03 доллара.[1]

---

[1] Эти цифры скорректированы с учетом инфляции, и в их основе лежат цены на жесткие диски, взятые с *www.littletechshoppe.com/ns1625/winchest.html*.

В известном смысле можно считать, что Excel фактически *становится меньше*!

Что же, в точности, имеется в виду под bloatware? Одно злобное определение[1] гласит, что это «программное обеспечение, предоставляющее незначительную функциональность при непропорционально большом объеме необходимой ему дисковой или оперативной памяти. Особенно характерно для обновлений приложений и ОС. Термин очень распространен в мире Windows/NT. Как и его причина».

Я думаю, что они просто ненавидят Windows. С нехваткой памяти я не сталкивался не меньше десяти лет, с тех пор как в Windows 386 появилась виртуальная память (1989 г.). А цена дисковой памяти снизилась до $0,0071 за мегабайт и продолжает падать, как овца, которая прыгнула с дерева, чтобы научиться летать.

Может быть, Линус Акерлунд это объяснит. На своей веб-странице он пишет: «Огромный недостаток этих раздутых программ состоит в том, что приходится загружать очень большую программу, даже когда надо выполнить одну маленькую функцию. Она поглощает всю память... машина работает неэффективно. Система кажется менее эффективной, чем это есть на самом деле, тогда как в этом нет никакой необходимости».[2]

Ай-яй-яй. Съедает всю память. Понятно. Ну, вообще-то не съедает. Начиная с Windows 1.0 в 1987 году операционная система загружает только те страницы, которые используются. Если исполняемый модуль имеет размер 15 Мбайт, а вы используете только тот код, который занимает страницы объемом 2 Мбайт, то с диска в оперативную память загрузятся только 2 Мбайт. На самом деле любая современная версия Windows автоматически перегруппирует эти страницы на жестком диске, чтобы они располагались последовательно, и тогда в следующий раз программа запустится еще быстрее.

И мне кажется, что никто не станет отрицать, что на современных сверхмощных и подешевевших компьютерах большая программа загружается все равно быстрее, чем загружалась маленькая программа пять лет назад. Так в чем проблема?

---

[1] См. *www.catb.org/~esr/jargon/*.

[2] Ekerlund, Linus (Линус Акерлунд) «Why I don't like bloatware: A perfectly normal rant» (Почему мне не нравится bloatware – совершенно обычная проповедь). Домашняя страница Линуса, 1998. См. *user.tninet.se/~uxm165t/bloatware.html*.

Подсказку нам даст программист, оставшийся анонимным. Оказывается, он потратил *часы* на то, чтобы препарировать небольшую утилиту Microsoft, и пришел в ярость, что она занимает целый мегабайт памяти. (На $0,0315 дисковой памяти по ценам в момент написания статьи.) По его мнению, программа должна быть на 95% меньше. Юмор в том, что препарированная им утилита – это RegClean, о которой вы, возможно, не слыхали. Эта программа просматривает реестр Windows, отыскивает в нем то, что не используется, и удаляет. Нужно иметь некоторые признаки навязчивого невроза, чтобы заботиться о чистке неиспользуемых элементов реестра. Поэтому я начинаю подозревать, что беспокойство по поводу bloatware относится скорее к проблемам душевного здоровья, чем к проблемам программирования.

В действительности есть множество веских причин, которыми можно оправдать снисходительное отношение к bloatware. Во-первых, если программисты не должны тревожиться о размере кода, то они быстрее выпустят готовый продукт. А это значит, что пользователи получат больше функций, а функции облегчают им жизнь (если их применять) и обычно не мешают (если их не применять). Если производитель нужной вам программы перед тем, как выпустить ее, сделает паузу и потратит два месяца на то, чтобы сделать код компактнее и сжать его вдвое, вы от этого выиграете незначительно. Конечно, если ваш диск почти всегда забит под завязку, то вы сможете в результате загрузить лишний mp3-файл группы «Дюран Дюран». А вот ваши убытки из-за необходимости ждать новой версии два лишних месяца действительно могут оказаться ощутимыми, а убытки софтверной компании от двух месяцев задержки продаж окажутся еще более существенными.

Многие разработчики программного обеспечения соблазняются древним правилом «80/20». Оно *кажется* очень разумным: 80% людей использует только 20% всех функций. Поэтому вы говорите себе, что достаточно реализовать 20% функций, чтобы продать 80% от максимально возможного количества экземпляров программы.

К сожалению, это не одни и те же 20%. Всем нужны *разные* группы функций. За последние десять лет мне пришлось слышать о *десятках* компаний, которые, твердо решившись не учиться на чужом опыте, пытались выпускать «облегченные» текстовые процессоры, реализующие всего 20% функций. Эта история так же стара, как PC. Чаще всего они дают свою программу журналисту для рецензирования, и журналист рецензирует ее, редактируя текст рецензии в этом новом текстовом процессоре, а потом

журналист пытается найти функцию «подсчет слов», которая ему необходима, потому что у большинства журналистов жесткие требования к количеству слов, а этой функции нет, потому что она попала в «80% никем не используемых», и в итоге журналист пишет статью, в которой пытается одновременно утверждать, что облегченные программы – это хорошо, раздутые программы – плохо, и «не могу пользоваться этой чертовой штукой, потому что она не считает слова».[1] Если бы всякий раз,[2] как это случается, я получал доллар, то был бы очень доволен.

Когда вы начинаете предлагать свой «облегченный» продукт и говорите всем, что «он занимает всего 1 Мбайт», все кажутся очень довольными. Потом они спрашивают, есть ли в нем некая необходимая *им* функция, а ее нет, и они не покупают ваш продукт.

Итог: выбрав стратегию 80/20, вы не сможете успешно продавать свою программу. Такова реальность. Эта стратегия так же стара, как сама программная индустрия, и она не оправдывает себя; удивляет лишь, сколько руководителей «быстрорастущих компаний» уверено в ее действенности.

Лучше всего об этом сказал Джейми Завински, обсуждая первоначальную версию Netscape, которая изменила мир.[3] «Как хотелось бы сказать, будь это правдой, что Mozilla [Netscape 1.0] огромен не из-за того, что в нем полно всякого бесполезного хлама. Mozilla огромен, потому что ваши потребности огромны. Ваши потребности огромны, потому что Интернет огромен. Есть множество маленьких тощеньких броузеров, которые, надо сказать, почти ничего не могут. Но создание ослепительного образца совершенства не было нашей целью, когда мы писали Mozilla.»

---

1 David Coursey (Дэвид Корси) «Want a cheap alternative to MS Office? Here's why you should try ThinkFree» (Дешевая альтернатива MS Office? Попробуйте ThinkFree), ZDNet, February 2, 2001. См. *wwwzdnet.com/anchordesk/stories/story/0,10738,2681437,00.html*.

2 Charles Bermant (Чарльз Берман) «Yeah Write», WashingtonPost.com, June 27, 1997. См. *washingtonpost.com/wp-srv/tech/reviews/finder/rev_1030.htm*.

3 Jamie Zawinski «easter eggs», *www.jwz.org/doc/easter-eggs.html*, 1998.

ГЛАВА СОРОКОВАЯ

# Пятое письмо о стратегии: экономика Open Source

12 ИЮНЯ 2002 ГОДА, СРЕДА

Во время учебы в колледже я посещал два экономических курса: макроэкономику и микроэкономику. В «макро» было полно теорий типа «низкая безработица порождает инфляцию», которые всегда плохо согласовывались с реальностью. Но «микро»-теории были интересными и полезными. Там были всякие идеи относительно соотношения спроса и предложения, которые действительно работали. Например, если ваш конкурент снижает цены, то спрос на ваш продукт будет снижаться, пока вы не приведете в соответствие свои цены.

Здесь я покажу, как одна из этих идей многое проясняет в работе некоторых известных компьютерных компаний. Попутно я обнаружил нечто интересное, касающееся ПО с открытым исходным кодом. Большинство компаний, тратящих большие деньги на разработку такого ПО, делают это потому, что для них это стратегически выгодно, а не потому, что они внезапно разуверились в капитализме и прониклись любовью к «свободе как к свободе слова».[1]

У каждого товара на рынке есть *заменитель* и *дополнитель* (*комплементарный ему*). Заменитель – это другой товар, который можно купить, если первый продукт окажется слишком дорог. Цыплята заменяют говяди-

[1] «Freedom-as-in-speech» имеет совершенно конкретное техническое значение у поборников бесплатного ПО (см. обсуждение на *www.gnu.org/philosophy/free-sw.html*). По существу там говорится, что можно делать с ПО что угодно, но это не значит, что за него не надо платить.

ну. Если вы выращиваете цыплят и цена на говядину растет, то вырастет спрос на цыплят, как и объем вашей торговли.

Дополняющий товар – это тот, который вы обычно покупаете вместе с другим товаром. Бензин и автомобили дополняют друг друга. Компьютерное «железо» – классическое дополнение операционных систем. А бэби-ситтеры – это дополнение ужина в хорошем ресторане. Если в маленьком городке пятизвездный ресторан делает в день св. Валентина специальное предложение «два по цене одного», то местные бэби-ситтеры удваивают свой тариф. (Фактически уже девятилетки вовлекаются в сферу обслуживания.)

При прочих равных условиях спрос на продукт растет, если цена на его дополнение снижается.

Я позволю себе повториться, потому что это важный момент. Спрос на продукт растет, если цена на его дополнение снижается. Например, если перелет в Майами становится дешевле, спрос на номера в гостиницах Майами растет – потому что больше людей летит в Майами и им нужны номера. Когда компьютеры становятся дешевле, больше людей покупает их, и всем им требуется операционная система, поэтому спрос на операционные системы растет, а значит, цена операционных систем может подняться.

В этот момент люди довольно часто пытаются мутить воду словами: «Ха! Но Linux *бесплатна*!» Разберемся. Во-первых, когда экономист говорит о цене, он рассматривает полную цену, включающую в себя такие трудноосязаемые вещи, как время, необходимое для установки, переобучения работников и преобразования имеющихся процедур. Все то, что мы любим называть «совокупной стоимостью владения».

Во-вторых, опираясь на такой аргумент, как бесплатность, эти проповедники решают, что законы экономики на них не действуют, потому что у них есть замечательный ноль, на который можно все помножить. Вот пример. Когда Slashdot спросил у разработчика Linux Моше Бара, будут ли новые ядра Linux совместимы с существующими драйверами устройств, он ответил, что в этом нет необходимости. «Закрытое программное обеспечение идет по цене от 50 до 200 долларов за строчку отлаженного кода. К ПО OpenSource эти цены не относятся».[1] И далее Моше заявляет, что не будет ничего страшного, если очередная версия ядра сделает устаревшими все существующие драйверы, потому что стоимость переписывания всех

[1] Интервью Моше Бара (Moshe Bar), данное им Slashdot 2 июня 2002, см. *http://interviews.slashdot.org/inter views/02/06/07/1255227.shtml?tid=156*.

имеющихся драйверов равна нулю. Это совершенно неверно. Фактически он утверждает, что потратить немного времени на то, чтобы сделать ядро обратно совместимым, эквивалентно тому, чтобы потратить огромное количество времени на переписывание всех драйверов, поскольку то и другое умножается на их «стоимость», которую он считает равной нулю. Это ошибка, основанная на внешнем впечатлении. Тысячи или миллионы часов работы программистов, которые нужны, чтобы переделать все существующие драйверы устройств, будут потрачены в ущерб чему-то другому. А пока эта работа не будет выполнена, Linux снова будет отставать на рынке из-за неполной поддержки имеющихся устройств. Не лучше ли было бы потратить все эти усилия с «нулевой стоимостью» на то, чтобы усовершенствовать Gnome? Или на поддержку новых устройств?

Отлаженный код *не является* бесплатным, находится он в чьей-то собственности или он открыт. Даже если за него не платят наличными, у него есть стоимость упущенных возможностей и стоимость времени. Объем таланта программистов-добровольцев, которых можно привлечь к работе над открытым кодом, ограничен, и все проекты с открытым исходным кодом конкурируют друг с другом за ограниченные ресурсы программирования, и только в самых привлекательных проектах желающих участвовать больше, чем можно реально привлечь. Короче, меня не очень убеждают люди, пытающиеся доказывать дикие с точки зрения экономики вещи относительно бесплатного ПО, потому что, насколько я понимаю, они просто получают ошибку деления на ноль.

Open source – не исключение из закона тяготения или законов экономики. Мы это видели на примере Eazel, ArsDigita, компании, ранее известной как VA Linux, и множества других продуктов. Но все же происходит нечто такое, что поняли по-настоящему очень немногие в open source: множество очень крупных открытых компаний, которые обязаны увеличивать капитал своих акционеров, инвестирует огромные деньги в поддержку производства ПО open source, обычно оплачивая работу над ним большим бригадам программистов. Это объяснимо с точки зрения комплементарности товаров.

Еще раз: спрос на продукт растет, когда цена дополняющего продукта падает. В принципе стратегический интерес компании состоит в том, чтобы цена товаров, комплементарных производимым ею самой, стала как можно ниже. Самой нижней теоретически возможной ценой является цена взаимозаменяемых товаров, которая формируется, когда есть группа конкурентов, предлагающих неотличимые один от другого товары. Поэтому:

**Если компанией руководят умные люди, то они стремятся, чтобы товары, дополнительные к тем, которые они производят, продавались по низкой цене взаимозаменяемых товаров.**

Если вам удастся это сделать, спрос на ваш продукт вырастет, и вы сможете назначить за него более высокую цену.

Когда в IBM проектировали архитектуру PC, то брали готовые комплектующие вместо того, чтобы разрабатывать специальные компоненты, и тщательно задокументировали интерфейсы между компонентами в (революционном) техническом руководстве IBM-PC. Зачем? Чтобы другие производители смогли вступить в игру. То, что соответствует интерфейсу, может быть использовано в PC. *Целью IBM было создать рынок взаимозаменяемых дополнительных устройств*, который является дополнительным к рынку PC, и они вполне с этим справились. В очень короткое время возникла тьма компаний, предлагающих платы памяти, жесткие диски, графические карты, принтеры и т. д. Дешевизна дополнительных устройств увеличивала спрос на PC.

Когда IBM лицензировала операционную систему PC-DOS у Microsoft, последняя очень постаралась, чтобы эта лицензия не была эксклюзивной. Благодаря этому Microsoft смогла лицензировать то же самое для Compaq и сотен других OEM, которые легально клонировали IBM-PC с помощью собственной документации IBM. *Целью Microsoft было создание рынка дешевых взаимозаменяемых PC.* Очень скоро сам PC стал товаром широкого потребления с постоянно падающими ценами, постоянно растущей мощностью и суровой маржей, делающей получение прибыли крайне трудным. Низкие цены, разумеется, увеличивают спрос. Растущий спрос на PC означал растущий спрос на дополняющий товар, MS-DOS. При прочих равных условиях, чем выше спрос на продукт, тем больше денег он вам приносит. Вот поэтому Билл Гейтс может купить Швецию, а вы нет.

В этом году Microsoft пытается повторить фокус: в их новой игровой приставке XBox используются обычные компоненты PC вместо заказных комплектующих. Идея была в том, что поскольку аппаратные компоненты дешевеют с каждым годом, то XBox могла бы сбить цены.[1] К сожалению, похоже, что стратегия обернулась против авторов. Очевидно, что цены на устройства PC уже снизились до минимальной цены товара на рынке, по-

[1] Takahashi, Dean (Дин Такахаши) «Opening the Xbox: Inside Microsoft's Plan to Unleash an Entertainment Revolution» (Открывая Xbox: тайные планы Microsoft сделать революцию в индустрии развлечений), Prima Lifestyles, 2002.

этому цена изготовления XBox уменьшается не так быстро, как хотелось бы Microsoft. Частью стратегии Microsoft в отношении XBox было использование DirectX, графической библиотеки, с помощью которой можно писать код, работающий на всевозможных видеочипах. Цель здесь в том, чтобы сделать видеочип взаимозаменяемым товаром и сбить на него цены, а тогда можно будет продавать много игр – вот где главный источник прибыли. А почему бы производителям видеочипов не попытаться каким-то образом сделать игры взаимозаменяемым товаром? Это *намного* труднее. Игра «Halo» продается с бешеным успехом, но у нее *нет* заменителей. Если вы собираетесь пойти в кинотеатр посмотреть «Звездные войны: атака клонов», то это не значит, что вас удовлетворил бы и фильм Вуди Аллена. Оба фильма могут быть замечательными, но они не вполне заменяют друг друга. Так кем бы вы предпочли быть – издателем игр или производителем видеочипов?

Старайтесь сделать взаимозаменяемыми товары, дополнительные к тем, которые вы производите.

Поняв эту стратегию, гораздо легче разобраться, почему многие коммерческие компании вкладывают большие средства в open source. Перейдем теперь к ним.

## *Заголовок: IBM тратит миллионы на разработку ПО Open Source*

Миф: они это делают, потому что Лу Герстнер прочел манифест GNU и решил, что не любит капитализм.

Реальность: они это делают, потому что IBM становится консалтинговой компанией в области ИТ. ИТ-консалтинг – это дополняющий товар к промышленному программному обеспечению. Поэтому IBM стремится сделать промышленное ПО дешевым взаимозаменяемым товаром, и лучший способ для этого – поддержка open source. И вот, их консалтинговое отделение добилось больших успехов с этой стратегией.

## *Заголовок: Netscape переводит свой броузер в Open Source*

Миф: они делают это для того, чтобы новозеландцы, сидя в интернет-кафе, писали для них бесплатный исходный код.

Реальность: они это делают, чтобы веб-броузер стал дешевым взаимозаменяемым продуктом.

Такова была стратегия Netscape *с самого начала*. Посмотрите на самый первый пресс-релиз Netscape:[1] броузер относится к классу «freeware». Netscape отдала броузер, собираясь делать деньги на серверах. Броузеры и серверы – классические дополняющие товары. Чем дешевле броузеры, тем больше серверов вы продадите. Это вполне подтвердилось в октябре 1994 года. (В Netscape были удивлены, когда в дверях появилась MCI (Microwave Communications, Inc.) и вывалила им такую кучу денег, что возможность делать деньги и на броузере тоже стала очевидной.[2] Их бизнес-план этого не требовал.)

Netscape выпустила Mozilla как open source потому, что это рассматривалось как возможность снизить стоимость разработки броузера. В результате они получали выгоды от предельно низкой рыночной цены дополнительного продукта, но с меньшими затратами.

Позднее AOL/Time Warner купила Netscape. Серверное программное обеспечение, которое должно было приносить доход благодаря дешевым броузерам, оказалось не очень прибыльным и было выкинуто. Есть ли теперь для AOL/Time Warner смысл продолжать инвестировать в open source?

AOL/Time Warner – это поставщик развлекательной продукции. Компании-поставщики развлечений служат дополнением ко всем типам платформ доставки развлечений, включая веб-броузеры. Стратегический интерес этого гигантского конгломерата заключается в том, чтобы сделать доставку развлечений – веб-броузеры – товаром, за который никто не сможет брать деньги.

Мои аргументы немного подпорчены тем, что Internet Explorer распространяется бесплатно. Microsoft тоже желала сделать веб-броузеры дешевым товаром, чтобы увеличить продажи операционных систем для настольных машин и серверов. Они сделали дополнительный шаг, выпустив набор компонент, из которых каждый мог собрать броузер. Neoplanet, AOL и Juno построили из этих компонент собственные броузеры. Но если IE бесплатен, какой смысл Netscape делать свой броузер «еще дешевле»? Это упреждающий шаг. Им нужно помешать Microsoft захватить полную монополию в веб-броузерах, даже бесплатных веб-броузерах, потому что иначе

---

[1] Netscape Communications News Release «Netscape Communications Offers New Network Navigator Free on the Internet», Netscape.com, 1999. См. *http://wp.netscape.com/newsref/pr/newsrelease1.html*.

[2] Netscape Communications News Release «MCI Selects Netscape Communications' Secure Software for New InternetMCI Service», Netscape.com, 1999. См. *http://wp.netscape.com/newsref/pr/newsrelease4.html*.

теоретически Microsoft получит возможность увеличить цену доступа к Интернету другими способами, например, подняв цену Windows.

(Мои аргументы даже еще слабее, поскольку вполне очевидно, что во времена Барксдейла в Netscape плохо представляли себе, что они делают.[1] Более вероятное объяснение того, что сделали в Netscape, заключается в том, что высшее руководство было технически неграмотно, и им не оставалось ничего иного, как согласиться с той схемой, которую предложили разработчики. Разработчики были хакерами, а не экономистами, и лишь случайно предложили схему, которая соответствовала их стратегии. Но пусть это останется предположением.)

## *Заголовок: Transmeta нанимает Линуса и платит ему за развитие Linux*

Миф: они сделали это ради рекламы. Кто бы иначе узнал о существовании Transmeta?

Реальность: Transmeta производит CPU. Естественным дополнением CPU является операционная система. Трансмета хочет, чтобы ОС была дешевым взаимозаменяемым товаром.

## *Заголовок: Sun и HP оплачивают Ximian работу над Gnome*

Миф: Sun и HP поддерживают разработку свободного программного обеспечения, потому что Базары нравятся им больше, чем Соборы.

Реальность: Sun и HP – производители аппаратного обеспечения. Они делают ящики. Чтобы зарабатывать деньги на настольных компьютерах, им нужно, чтобы оконные системы, являющиеся дополнением настольных компьютеров, были дешевым массовым товаром. Почему бы им на те деньги, которые они платят Ximian, не разработать собственную оконную систему? Они пробовали это сделать (у Sun есть NeWS, а у HP – New Wave), но по сути своей это компании, производящие аппаратуру и не достигшие мастерства в разработке ПО, а требующиеся им оконные системы должны быть *дешевым массовым товаром*, а не корпоративной собственностью,

[1] Ferguson, Charles (Чарльз Фергюсон) «High Stakes, No Prisoners: A Winner's Tale of Greed and Glory in the Internet Wars» (Ставки высоки, пленных не брать: рассказ победителя о жадности и славе в интернет-войнах), Crown Business, 1999.

за которую нужно платить. Поэтому они наняли для этой работы отличных парней из Ximian – по той же причине, по которой Sun купила StarOffice и передала его в open source: чтобы сделать ПО дешевым массовым товаром и больше заработать на аппаратуре.

## *Заголовок: Sun разрабатывает Java; новая система «байт-кода» означает выполнение кода на любой платформе*

Идея байт-кода не нова. Программисты всегда старались, чтобы их код могло выполнять как можно больше машин. (Вот так снижается цена товара, дополнительного к вашему.) У Microsoft давно был свой компилятор *p*-кода и переносимый слой оконной системы, благодаря чему Excel работала под Mac OS, Windows и OS/2, на чипах Motorola, Intel, Alpha, MIPS и PowerPC. У Quark есть слой, выполняющий код Macintosh под Windows. Язык программирования C лучше всего описать как аппаратно-независимый язык ассемблера. Так что идея не нова для программистов.

Если ваши программы могут выполняться всюду, это способствует превращению «железа» в дешевый взаимозаменяемый товар. Падение цен на аппаратуру влечет расширение рынка, которое вызывает увеличение спроса на ПО (при этом у покупателей остается больше денег на покупку программ, которые в результате можно сделать дороже.)

Энтузиазм Sun по поводу WORA (Write Once, Run Anywhere) вызывает *недоумение*, потому что Sun – это производитель аппаратуры. Превращение аппаратной части в дешевый взаимозаменяемый товар – последнее, в чем они должны быть заинтересованы.

Все, приехали!

Sun – это бешеная собака компьютерной индустрии. Ослепнув от неистового страха и ненависти к Microsoft, они выбирают стратегии, основываясь больше на злости, чем на собственных интересах. Вот две стратегии Sun:

1. Сделать программное обеспечение взаимозаменяемым товаром, продвигая и разрабатывая бесплатное ПО (StarOffice, Linux, Apache, Gnome и т. д.).
2. Сделать аппаратное обеспечение взаимозаменяемым товаром путем продвижения Java с его архитектурой байт-кодов и WORA. Хорошо, вопрос к Sun: когда закончится музыка, на какой стул вы сядете? В от-

сутствие преимуществ прав собственности как в аппаратном, так и в программном обеспечении вам придется принять низкие цены взаимозаменяемых товаров, которые едва ли окупят стоимость дешевых фабрик в Гвадалахаре, не то что ваших роскошных офисов в Кремниевой долине.

«Но, Джоэл! – говорит Джаред. – Sun пытается сделать дешевую операционную систему, как и Transmeta, а не аппаратуру». Возможно, но байткод Java также удешевляет аппаратную часть, а это довольно чувствительный побочный ущерб.

В этих примерах важно обратить внимание на то, что программному обеспечению легко повлиять на превращение аппаратуры во взаимозаменяемый товар (достаточно написать небольшой слой аппаратных абстракций, как HAL в Windows NT, а это совсем немного кода), но аппаратуре повлиять на взаимозаменяемость программ крайне трудно. Программное обеспечение не является взаимозаменяемым, как убеждается группа маркетинга StarOffice. Даже если цена нулевая, стоимость перехода с Microsoft Office не равна нулю. Пока стоимость перехода на другой продукт не станет равна нулю, офисные приложения не станут в полном смысле дешевыми взаимозаменяемыми продуктами. Даже мелкие различия могут сделать болезненным переход с одного пакета на другой. Хотя в Mozilla есть все нужные мне функции, и я с удовольствием пользовался бы им, чтобы избавиться от этой безумно раздражающей всплывающей рекламы, но я слишком привык к тому, чтобы переходить в окно адреса по Alt+D. Так что извините. Одна крошечная деталь, и статус взаимозаменяемого товара утрачен. Но я вытаскивал жесткие диски из компьютеров IBM, ставил их в компьютеры Dell, и система после этого отлично запускалась точно так же, как если бы диск стоял в старой машине.

Амос Михельсон (Amos Michelson), глава Creo,[1] рассказывал мне, что у него в компании все служащие должны пройти курс дисциплины, которую он назвал «экономическим мышлением». Замечательная мысль. Даже простые понятия из основ микроэкономики очень способствуют пониманию некоторых происходящих сегодня изменений.

[1] См. *www.creo.com/*.

ГЛАВА СОРОК ПЕРВАЯ

# Неделя буйства закона Мерфи

25 января 2003 года, суббота

## *Часть первая*

Сервер Linux, на котором размещалось наше хранилище CVS (весь наш исходный код), упал. Невелика беда: он автоматически копируется (с помощью rsync) на удаленную машину. Сжатие и пересылка скопированных данных занимают несколько часов. Обнаруживается, однако, что мы забыли задать для rsync параметр, с помощью которого убираются помеченные для удаления файлы, поэтому наше зеркальное отображение не идеально: оно содержит удаленные файлы. Их надо убирать вручную.

Покончив с этим, я решаю загрузить себе целиком все дерево исходного кода и сравнить с тем, что у меня уже есть, проведя таким образом окончательную проверку. Но у меня на жестком диске моего лэптопа недостаточно места. Настало время для апгрейда. Заказываю диск 60 Гбайт для лэптопа и коннектор PCMCIA/жесткий диск, с помощью которого предполагается осуществить клонирование старого жесткого диска на новый. Эта процедура занимает около шести часов и аварийно прерывается примерно на середине работы с указанием «запустить scandisk». Это занимает еще пару часов. Снова начинаем копирование. Еще шесть часов. Посередине копирования снова отказ. Но теперь пришел конец моему *исходному* диску вместе со всем, что на нем было. После двух часов возни с ним и попыток вставить его в другой компьютер и т. п. становится ясно, что он действительно потерян.

Плохо, но тоже не смертельно: у нас есть ежедневные резервные копии (NetBackup Pro). Вставляю в лэптоп новый диск 60 Гбайт, форматирую его и устанавливаю Windows XP Pro. Даю NetBackup Pro команду восстановить машину в состояние перед аварией. Потеряю результаты дневной работы, но я почти ничего за этот день и не сделал. Потеряна почта за день, поэтому, если вы мне писали на этой неделе и я вам не ответил, напишите снова.

NetBackup Pro работает несколько часов. Ухожу домой, не дождавшись завершения. Утром обнаруживается, что система мертва и даже не загружается. Делаю предположение, что я пытался восстановить образ, снятый с Win2K, на машину с XP Pro. Начинаю сначала, на этот раз установив Win2K (форматирование жесткого диска: один час; установка Win2K: один час; потом установка клиента NetBackup Pro). Снова запускаю восстановление. Пятью часами позже оно выполнено еще только наполовину, и я ухожу домой.

На следующее утро система *не совсем* загружается, выводит голубой экран, но полчаса в Safe Mode, и я успешно запускаю машину. И, о радость, все восстановлено, за исключением того, что недоступны несколько файлов, которые я позволил Windows зашифровать (с помощью EFS). Это как-то связано с открытыми ключами и сертификатами. Похоже, что при восстановлении зашифрованного файла его нельзя прочесть. Решения этой проблемы я пока не нашел. Если скажете мне, что надо сделать, буду вам век благодарен. (26 января. Решил проблему после нескольких часов выдирания волос на голове.)

### *Извлеченные уроки*

Это не первый раз, когда отказ жесткого диска приводит к ряду других проблем и оканчивается потерей нескольких дней работы. Обратите внимание, что у меня была вполне приличная стратегия резервного копирования: все копировалось ежедневно на удаленную машину. Пожалуй, это уже третий раз, когда авария жесткого диска ведет к серии неудач и потере времени. Вывод: резервного копирования недостаточно. Отныне будет использоваться RAID с зеркальным копированием. Если сдохнет один диск, я потрачу 15 минут, чтобы вставить новый, и продолжу ровно с того места, где остановился. Новая политика: все компьютеры в Fog Creek, кроме лэптопов, будут иметь зеркальный RAID.

## *Часть вторая*

Вы заметили, что у нас не работал веб-сервер? В пятницу, около полудня, пожар на местной станции Verizon вывел из строя все наши телефонные линии и подключение к Интернету. Verizon восстановил телефонную связь через пару часов, но с T1 оказалось немного сложнее. Мы приобрели T1 у Savvis, а те, в свою очередь, взяли местную линию у MCI, которую теперь зовут WorldCom, но WorldCom, конечно, сама *не занимается* никакими местными линиями – вот еще, руки *пачкать*, – они просто покупают местную линию у Verizon.

В итоге с полудня пятницы и до полуночи субботы мы с Майклом, сменяя друг друга, как борцы на ковре, звоним примерно каждый час в Savvis и выясняем, как идут дела. Мы давим на Savvis, а те время от времени давят на WorldCom, где решили, что во всем виновата какая-то распределенная атака на SQL Server, и потому мало внимания обращают на Savvis, которые не говорят нам, что WorldCom их игнорирует, и мы *снова* давим на Savvis, а *они* снова давят на WorldCom, и где-то с третьего раза WorldCom соглашается обратиться в Verizon, откуда присылают техника, который все и исправляет. В самом деле, это какая-то *цепочка* давлений. Когда в предыдущий раз Savvis оставили нас без T1 на сутки, техническая проблема была относительно проста и могла быть обнаружена и исправлена в течение нескольких минут, если бы тут не участвовало столько дурацких компаний.

## *Извлеченные уроки*

Когда вы покупаете услугу у компании, просто передавшей кому-то осуществление этой услуги на один уровень ниже, приличное обслуживание клиентов маловероятно. Когда таких уровней передачи два, оно практически невозможно. Хоть я и не люблю поощрять местных монополистов-связистов, но хуже непосредственных отношений с местными связистами могут быть только отношения еще с одной идиотской бюрократизированной компанией, у которой самой нет выбора, кроме как работать с местной телекоммуникационной компанией. Спасибо, в нашем новом офисе связь будет осуществляться через Verizon DSL.

Между прочим, никто и не заметил бы этого перебоя, если бы Dell вовремя доставила нам наш сервер. Мы должны были еще месяц назад начать работать в прекрасной высоконадежной среде Peer 1, разместив свой сер-

вер в защищенном помещении. Я уже сказал, что у меня лихорадка? Я всегда заболеваю, когда возникают неприятности.

## *Часть третья*

В тысячный раз отопление на четвертом этаже особняка Fog Creek перестало действовать. Тепло поступает от труб с горячей водой, проходящих внутри стен. Эти трубы замерзли. Как это могло случиться? Дело в том, что на прошлой неделе отключилась печь, потому что ее устанавливал какой-то идиот, наверное, не имеющий лицензии, и он поставил 8-метровый *горизонтальный* кусок дымохода, из-за чего нет тяги, уже был госпитализирован один жилец и десятки раз выключалась печка. Наконец, кто-то в отопительной компании предложил поставить усилитель тяги, чтобы создать принудительное движение воздуха, что они и сделали, но уже после того, как замерзли трубы. Разумеется, трубы были плохо изолированы, потому что их ставил какой-то другой некомпетентный нью-йоркский строитель, но это было бы нестрашно, если бы работала печка.

## *Извлеченные уроки*

Слабые системы могут казаться вполне безопасными, пока не сломаются какие-нибудь смежные системы. Люди с аллергией *и* больным позвоночником могут месяцами не испытывать неприятностей, связанных с одним или другим заболеванием, пока внезапно сенная лихорадка не заставит их чихнуть так, что у них выскочит позвонок. Это можно *постоянно* наблюдать в системном администрировании. Постарайтесь *исправлять все проблемы сразу*. Установите RAID на *все* компьютеры *и* делайте резервные копии, не пользуйтесь EFS и всегда ставьте жесткие диски побольше, чтобы не пришлось останавливать работу для их замены, и тщательно проверяйте параметры командной строки rsync. Установите воздуходувку *и* изолируйте трубы. Переместите свои важные серверы в защищенную среду и переведите офисный T1 на Verizon.

ГЛАВА СОРОК ВТОРАЯ

# Как Microsoft проиграла войну API

13 июня 2004 года

Вокруг то и дело говорят: «Microsoft кончена. Как только Linux заполонит десктопы, а настольные приложения сменятся веб-приложениями, могущественная империя рухнет».

Отчасти верно, что Linux – это большая угроза для Microsoft, но предсказания кончины компании из Редмонда весьма и весьма преждевременны. У Microsoft несметные деньги в банке, и она по-прежнему приносит огромную прибыль. Если ей суждено падение, то оно будет очень долгим. Она может лет десять делать одну ошибку за другой, прежде чем почувствует хоть какую-то опасность, и как знать – может быть, в последнюю минуту они придумают себе какое-нибудь перевоплощение. Так что не торопитесь списывать их со счетов. В начале 1990-х все думали, что IBM пришел полный конец: мэйнфреймы ушли в историю! В то время Роберт Крингли (Robert X. Cringely) предсказывал, что эра больших машин закончится 1 января 2000 года, когда выйдут из строя все приложения, написанные на COBOL, и вместо того чтобы исправлять эти приложения, исходный код которых, вероятно, давно утерян, все просто перепишут их для платформ клиент-сервер.

И что? Мэйнфреймы остались, 1 января 2000 года ничего не произошло, а IBM нашла себя вновь в качестве большой технологической консалтинговой компании, попутно изготавливающей дешевые пластмассовые телефоны. Поэтому тот, кто экстраполирует по нескольким точкам приближение конца Microsoft, впадает в очень большое преувеличение.

Однако есть другое явление, не до конца понятое и проходящее в значительной мере незамеченным: главный бриллиант в короне Microsoft,

а именно Windows API, потерян. Краеугольный камень власти Microsoft и исключительно прибыльных монополий Windows и Office, обеспечивающих практически все доходы Microsoft, которые покрывают убытки или мизерную прибыль множества других линий продуктов, Windows API больше не представляет особого интереса для разработчиков. Курица, несущая золотые яйца, еще не умерла, но она смертельно больна, чего никто пока не замечает.

Теперь позвольте извиниться за напыщенность и помпезность предыдущего абзаца. Я начал выражаться, как штатные авторы технических газетенок, которые все пишут и пишут о стратегически важном активе Microsoft, Windows API. Мне понадобится несколько страниц, чтобы объяснить, что я имею в виду, и привести свои аргументы. Пожалуйста, не спешите делать выводы, пока я не объясню, о чем говорю. Я должен объяснить, что такое Windows API и почему это самый важный стратегический актив Microsoft, почему он утрачен и какие долгосрочные последствия это может иметь. И поскольку я говорю о крупных тенденциях, мне придется преувеличивать и обобщать.

## *Разработчики, разработчики, разработчики, разработчики*

Помните определение операционной системы? Это такая штука, которая управляет ресурсами компьютера, чтобы могли выполняться прикладные программы. Люди не очень интересуются операционными системами: они интересуются прикладными программами, которые могут работать благодаря операционным системам. Текстовые процессоры. Instant Messaging. Электронная почта. Бухгалтерия. Веб-сайты с фотографиями Пэрис Хилтон. Сама по себе операционная система не очень полезна. Операционные системы покупают ради того, чтобы выполнять в них полезные программы. И поэтому самая полезная операционная система – это та, для которой есть больше всего полезных приложений.

Из этого логически следует, что самое главное для того, кто пытается продавать операционные системы, – это побудить разработчиков делать программы для этих операционных систем. По этой причине Стив Балмер и прыгал по сцене с криками «разработчики, разработчики, разработчики, разработчики».[1] Это настолько важно для Microsoft, что единственная при-

[1] См. *www.ntk.net/ballmer/mirrors.html*.

чина, по которой они *не раздают* бесплатно средства разработки для Windows, – это опасение перекрыть кислород конкурирующим с ними производителям средств разработки (тем, что еще остались), потому что наличие разнообразия в средствах разработки для некоторой платформы делает ее намного привлекательнее для разработчиков. Но они действительно *хотят* отдавать инструменты разработки бесплатно. Участвуя в их программе Empower ISV,[1] можно получить пять полных комплектов MSDN Universal (их называют еще «практически все продукты Microsoft, кроме Flight Simulator») примерно за $375. Компиляторы командной строки для языков .NET поставляются вместе с бесплатной средой выполнения .NET также бесплатно. Компилятор C++ теперь бесплатен.[2] Делается все, чтобы поощрить разработчиков к созданию программ для платформы .NET, и сдержанность проявляется только для того, чтобы уцелели такие компании, как Borland.

## *Почему Apple и Sun не могут продавать компьютеры*

Ну, это, конечно, чепуха. Конечно, Apple и Sun могут продавать компьютеры, но не на двух самых привлекательных рынках: корпоративных десктопов и домашних компьютеров. У Apple по-прежнему однозначные числа процентов доли рынка, а единственные люди, у которых на рабочих столах стоят Sun, работают в Sun. (Пожалуйста, поймите, что я рассказываю о крупных тенденциях, поэтому, если я говорю «никто», я имею в виду «меньше 10 000 000 человек».)

Почему? Потому что компьютеры Apple и Sun не выполняют программы Windows, а если и делают это, то в каком-нибудь жадном до ресурсов режиме эмуляции, который работает не слишком здорово. Напомню, что люди покупают компьютеры ради приложений, которые на них выполняются, а хороших настольных приложений для Windows настолько больше, чем для Mac OS, что быть пользователем Макинтоша очень трудно.[3]

Вот почему Windows API оказывается таким важным активом для Microsoft.

(О да, я знаю, что 2,3% всех владельцев компьютеров в мире, которые работают на Макинтошах, сейчас прогревают свои почтовые программы,

---

1 См. *members.microsoft.com/partner/competency/isvcomp/empower/default.aspx*.

2 См. *msdn.microsoft.com/visualc/vctoolkit2003/*.

3 Правда, теперь на Макинтошах уже запускают Windows XP. Так что Microsoft, похоже, крепко держит позицию. – *Примеч. ред.*

чтобы отправить убийственные письма с рассказами о том, как они любят свои Маки. Повторяю: я говорю о крупных тенденциях и обобщаю, поэтому не тратьте зря время. Я знаю, что вы любите свои Маки. Я знаю, что на нем есть все программы, которые нужны *вам*. Я люблю вас, вы прелесть, но вас всего 2,3%, и эта статья не о вас.)

### *Что это такое – «API»?*

Когда программист пишет программу, скажем, текстовый процессор, и ему надо вывести меню или записать файл, то он должен прибегнуть к средствам операционной системы, обратившись для этого к весьма специфическому набору вызовов функций, который в каждой операционной системе свой. Эти вызовы функций называются API: это интерфейс, который операционная система (например, Windows) предоставляет разработчикам приложений – программистам, создающим текстовые процессоры, электронные таблицы или что-то еще. Это набор из тысяч и тысяч детальных и подробных функций и подпрограмм, которыми могут пользоваться программисты и которые делают всякие интересные вещи: показывают меню, читают и пишут файлы или более экзотические вещи, например пишут указанную дату по-сербски, или очень сложные вещи, например, отображают веб-страницу в окне. Если программа основывается на вызовах API для Windows, то она не будет работать в Linux, у которой другие вызовы API. Иногда они выполняют примерно одинаковые вещи. Это одна из важных причин, по которым Windows-программы не работают под Linux. Если вы хотите, чтобы Windows-программа работала под Linux, вам придется реализовать в ней весь Windows API,[1] состоящий из тысяч сложных функций, а это почти такая же большая работа, как реализация самой Windows – то, что заняло у Microsoft тысячи человеко-лет. А если вы сделаете какую-нибудь маленькую ошибку или пропустите одну функцию, необходимую приложению, то это приложение аварийно завершится.

## *Две силы в Microsoft*

Внутри Microsoft есть две противостоящие силы, которые я несколько иронически назову лагерем Рэймонда Чена и лагерем MSDN Magazine.

[1] См., например, Wine – попытку open source сделать именно это (*www.winehq.com/*).

Рэймонд Чен – разработчик в команде Microsoft Windows. Он там с 1992 года, и в его веблоге «старое новое»[1] полным-полно подробных технических рассказов о том, почему разные вещи сделаны в Windows так, а не иначе, и даже у глупых на первый взгляд решений оказываются веские причины.

Самое сильное впечатление в блоге Рэймонда оставляют истории о невероятных усилиях, годами предпринимавшихся командой Windows для обеспечения обратной совместимости:

> *Взгляните с точки зрения покупателя на такой сценарий. Вы купили программы X, Y и Z. Затем вы перешли на Windows XP. Теперь в вашем компьютере время от времени возникает сбой, а программа Z вообще не работает. Вы скажете своим друзьям: «Не переходите на Windows XP. Она падает случайным образом и не совместима с программой Z». Станете ли вы глубоко разбираться со своей системой, чтобы выяснить, что сбои возникают из-за программы X, а программа Z не работает, потому что использует недокументированные сообщения окна? Нет, конечно. Вы просто вернете коробку Windows XP и потребуете назад деньги. (Программы X, Y и Z вы купили несколько месяцев назад. Вы уже не сможете получить за них деньги, положенные при возврате в течение 30 дней. Вернуть вы сможете только Windows XP.)*

Впервые я услышал об этом от одного из разработчиков популярной игры SimCity, который сказал мне, что в его приложении была критическая ошибка: она пыталась занять память сразу после того, как освобождала ее, – большое табу, нарушение которого *случайно* оставалось безнаказанным в DOS, но непреодолимое в Windows, где освободившаяся память может быть тут же занята другим приложением. Тестеры из команды Windows проверяли, нормально ли работают различные популярные приложения, и оказалось, что SimCity регулярно «падает». Они сообщили об этом разработчикам Windows, которые дизассемблировали SimCity, оттрассировали ее в отладчике, нашли ошибку и добавили специальный код, который проверял, не запущена ли SimCity, и если оказывалось, что да, то он запускал программу выделения памяти в особом режиме, разрешавшем использование только что освобожденной памяти.

---

[1] Raymond Chen «The Old New Thing». См. *weblogs.asp.net/oldnewthing/*.

Этот случай не был уникальным. Бригада тестирования Windows огромна, и одна из главных ее обязанностей состоит в том, чтобы гарантировать, что каждый может, не опасаясь, установить следующую версию операционной системы, и какие бы приложения ни были у него установлены, они сохранят работоспособность, даже если ведут себя не вполне корректно (используют недокументированные функции или ошибочное поведение Windows версии *n*, исправленное в Windows версии *n*+1). И действительно, если зайти в раздел AppCompatibility реестра Windows, там можно найти целый список приложений, с которыми Windows обращается по-особому, эмулируя разные старые ошибки и странности поведения, чтобы эти приложения могли работать. Рэймонд Чен пишет: «Особенно я возмущаюсь, когда Microsoft обвиняется в злонамеренном нарушении работы приложений, происходящем вследствие перехода к более новой версии ОС. Если оказывалось, что какое-то приложение не могло работать в Windows 95, я воспринимал это как личное поражение. Я провел много бессонных ночей, исправляя ошибки в чужих программах, чтобы они могли беспрепятственно работать в Windows 95».

Многие разработчики и инженеры не согласны с таким стилем работы. Если приложение ведет себя нехорошо или пользуется недокументированными функциями, то оно, как они считают, должно перестать работать при обновлении ОС. В давние времена к этому лагерю принадлежали разработчики ОС для Макинтошей (хотя со временем Apple стала прилагать гораздо больше усилий для обеспечения обратной совместимости). По этой причине очень немногие из первых приложений для Макинтошей продолжают работать и сейчас. И если создавшая такое некорректное приложение компания оказалась не у дел (а это случилось с большинством из них), то что ж, таковы суровые законы жизни.

И наоборот: у меня есть приложения DOS, которые я написал в 1983 году для самой первой IBM-PC, которые до сих пор работают без проблем благодаря лагерю Рэймонда Чена в Microsoft. Я, конечно, знаю, что дело не только в Рэймонде – это вообще *модус операнди* основной бригады Windows API. Но Рэймонд больше других рассказал об этом сообществу на своем отличном сайте, поэтому я назвал этот лагерь его именем.

Это один лагерь. Другой лагерь я назову лагерем MSDN Magazine – в честь журнала для разработчиков, в котором очень много замечательных статей о том, как можно создать себе неприятности, используя в своем программном обеспечении эзотерические комбинации продуктов Microsoft. Лагерь MSDN Magazine постоянно пытается убедить вас применять

новые и сложные внешние технологии типа COM+, MSMQ, MSDE, Microsoft Office, Internet Explorer и их компоненты, MSXML, DirectX (обязательно последнюю версию), Windows Media Player и Sharepoint (Sharepoint! – то, чего нет *ни у кого*) – подлинное прикрытие *внешних зависимостей*, каждая из которых создаст вам очень большие проблемы, если вы поставите свое приложение заплатившему деньги клиенту, а оно откажется работать. На языке технических специалистов это называется «кошмаром DLL». Если это работает здесь, то почему оно не работает там?

Лагерь Рэймонда Чена считает, что надо облегчать жизнь разработчикам, позволяя им писать программы, выполняемые всюду (т. е. на любой Windows-машине). Лагерь MSDN Magazine считает, что надо облегчать жизнь разработчикам, давая им мощные фрагменты кода, которыми можно пользоваться, если смириться с крайне сложным развертыванием и проблемами установки, не говоря уже о необходимости длительного обучения. Лагерь Рэймонда Чена озабочен консолидацией. Пожалуйста, давайте не будем ухудшать ситуацию, просто постараемся, чтобы то, что у нас уже есть, *по-прежнему работало*. Лагерь MSDN Magazine стремится по-прежнему выдавать на гора новые технологии огромного размера, за которыми никто не поспевает.

И вот почему это важно.

## *Обратная совместимость перестала быть священной коровой Microsoft*

Внутри Microsoft победу одержал лагерь MSDN Magazine. Его первый крупный успех – отсутствие обратной совместимости Visual Basic .NET с VB 6.0. Это было буквально впервые на моей памяти, когда при покупке новой версии продукта Microsoft оказалось, что прежние данные (т. е. код, написанный на VB6) нельзя было импортировать в точности и без проблем. Впервые обновленная версия продукта Microsoft не считалась с трудами пользователей в предыдущей версии продукта.

И *впечатление* было такое, что небо от этого не упало на землю – во всяком случае в Microsoft. Разработчики VB6 были готовы к борьбе, но их число все равно уменьшалось, потому что большинство из них были корпоративными разработчиками, а все они переходили на разработку веб-приложений. Подлинный отдаленный ущерб остался незамеченным.

Одержав такую крупную победу, лагерь MSDN Magazine возобладал. Внезапно стало дозволено менять порядок вещей. IIS 6.0 вышел с новой

моделью потоков, с которой не мог работать ряд старых приложений. Я был поражен, когда обнаружил, что у некоторых наших клиентов с Windows Server 2003 возникли проблемы с запуском FogBUGZ. Потом версия .NET 1.1 оказалась не вполне обратно совместимой с версией 1.0. Выпустив джина из бутылки, команда ОС вошла во вкус и решила, что вместо дополнения Windows API новыми функциями она полностью заменит его. Нам сообщают, что вместо Win32 мы должны готовиться к WinFX – новому поколению Windows API.[1] Совершенно новому. Основанному на .NET с управляемым кодом. XAML. Avalon. Да, я допускаю, что это будет намного лучше Win32. Но это не обновление,это разрыв с прошлым.

Сторонние разработчики, и раньше не слишком довольные сложностью разработки для Windows, стали *массами* покидать платформу Microsoft и теперь ведут разработки для Интернета. Пол Грэм (Paul Graham), создавший Yahoo! Stores на заре бума доткомов, красноречиво это подытожил: «У начинающих компаний все больше причин писать веб-приложения, поскольку создание приложений для рабочих мест стало гораздо менее интересным. Писать приложения для десктопов теперь придется на условиях Microsoft, вызывая их API и обходя ошибки в их ОС. И если вам удастся написать что-то успешное, то может оказаться, что вы просто выполняли маркетинговое исследование для Microsoft».[2]

Microsoft стала слишком велика, в ней слишком много разработчиков, и они слишком привыкли получать доходы от обновления версий, поэтому они вдруг решили, что изобретение *всего* заново *не* явится слишком большим проектом. Черт возьми, мы можем сделать это дважды. Старая Microsoft, Microsoft Рэймонда Чена, могла бы реализовать такие вещи, как Avalon, новую графическую систему, в виде ряда DLL, которые могли бы выполняться на любой версии Windows и объединяться с теми приложениями, которым они нужны. Технических препятствий для этого нет. Но Microsoft хочет, чтобы у вас появились причины купить Longhorn, и то, что они пытаются сделать, – это полная трансформация, аналогичная той, которая произошла, когда Windows сменила DOS. Беда в том, что Longhorn не сильно превосходит Windows XP – далеко не в той мере, в какой Windows превосходила DOS. Возможно, она не окажется настолько привлека-

---

1 Mark Driver (Марк Драйвер) «Microsoft WinFX Accelerates Need for .NET Adoption» (Microsoft WinFX ускоряет необходимость принятия .NET), Gartner Research, November 3, 2003. См. *www.gartner.com/DisplayDocument?doc_cd=118261*.

2 Paul Graham (Пол Грэм) «The Other Road Ahead» (Впереди другой путь), September 2001. См. *www.paulgraham.com/road.html*.

тельной, чтобы побудить людей покупать новые компьютеры и приложения, как они это делали для Windows. А может быть, и окажется, во всяком случае Microsoft обязательно приложит для этого усилия. Но то, что я видел до сих пор, выглядит не очень убедительно. Во многих случаях Microsoft сделала неверные ставки. Например, WinFS, рекламируемая как средство организации поиска путем превращения файловой системы в реляционную базу данных, игнорирует то, что действительный способ сделать поиск работающим – это заставить поиск работать.[1] Не требуйте, чтобы я вводил метаданные для всех файлов, в которых я могу делать поиск с помощью языка запросов. Сделайте одолжение: поищите – и быстро – на этом чертовом диске строку, которую я напечатал, с помощью полнотекстового указателя и прочих технологий, которыми надоедали в 1973 г.

## *Автоматические коробки передач одерживают верх*

Не поймите меня неправильно; я думаю, что .NET – замечательная среда разработки, а Avalon вместе с XAML – огромный шаг вперед по сравнению с прежним способом написания приложений GUI для Windows. Самое большое преимущество .NET заключается в автоматическом управлении памятью.

Многие из нас считали в 1990-х годах, что между процедурным и объектно-ориентированным программированием развернется большое сражение, и мы полагали, что ООП резко повысит производительность труда программистов. И я так считал. Некоторые до сих пор так считают. Оказывается, мы ошибались. ООП – удобная штука, но это не тот ускоритель труда, который был обещан. *Подлинный* существенный прирост продуктивности программирования был достигнут за счет языков, которые автоматически управляли памятью. Это может осуществляться с помощью счетчиков ссылок или сборки мусора; такими языками могут быть Java, Lisp, Visual Basic (даже 1.0), Smalltalk или любой язык сценариев. Если язык программирования позволяет занять участок памяти и не думать о том, как он будет освобождаться, когда перестанет быть нужен, значит, этот язык управляет памятью автоматически, и работа с ним будет гораздо эффективнее, чем с языком, в котором управление памятью целиком перекладывается на программиста. Если вы слышите, как кто-то хвастается, с каким

[1] John Udel, «Questions about Longhorn, part 1: WinFS», iDiscuss: Jon Udell's Weblog, InfoWorld, June 2, 2004. См. *weblog.infoworld.com/udell/2004/06/02.html#a1012*.

эффективным языком он работает, то скорее всего эта эффективность по большей части проистекает из автоматического управления памятью, даже если он сам объясняет ее чем-то иным.

### *Для чего нужно управление памятью?*

Почему автоматическое управление памятью так повышает производительность труда программиста?

1. Потому что можно написать `f(g(x))`, не беспокоясь о том, как освободить значение, возвращенное `g`, а это означает, что можно пользоваться функциями, которые возвращают замысловатые сложные типы данных, и функциями, которые преобразуют замысловатые сложные типы данных, что позволит работать на более высоком уровне абстракции.
2. Поскольку не придется тратить время на создание кода для освобождения памяти или обнаружения утечек памяти.
3. Потому что не надо тщательно координировать точки выхода из собственных функций, чтобы правильно зачищать память.

Поклонники гоночных автомобилей, вероятно, ответят мне разгневанными письмами, но мой опыт показывает, что только в одном случае, а именно при обычной езде, автоматическая коробка передач превосходит ручную. То же и в разработке программного обеспечения: почти всегда автоматическое управление памятью превосходит ручное и приводит к гораздо более высокой продуктивности программирования.

В первые годы после выхода Windows Microsoft предлагала разработчикам настольных приложений два пути: писать код C, который непосредственно вызывает Windows API, и самостоятельно управлять памятью, либо использовать Visual Basic, который распоряжался памятью вместо вас. Это те две среды разработки, в которых лично я больше всего работал за последние 13 лет, я знаю их вдоль и поперек и пришел к заключению, что Visual Basic *существенно* более продуктивен. Часто я писал *один и тот же код* отдельно на C++ с обращением к Windows API и отдельно на Visual Basic, и C++ всегда требовал в три-четыре раза больше работы. Почему? Из-за управления памятью. Проще всего увидеть это, посмотрев документацию по любой функции Windows API, которая возвращает строку. Обратите внимание, как многословно там рассказывается, кто должен выделять память для строки и как вести диалог по поводу необходимого объема памяти.

Обычно приходится вызывать функцию *дважды*: сначала вызывать, сообщив, что ей выделено ноль байт памяти, на что она отвечает, что этого недостаточно, и любезно подсказывает, сколько памяти необходимо зарезервировать. Это если вам повезло и вы не вызываете функцию, которая возвращает *список строк* или некую структуру переменного размера. Во всяком случае, такие простые действия, как открыть файл, записать в него строку и закрыть его, написанные через непосредственные обращения к Windows API, могут занять страницу кода. В Visual Basic аналогичные операции могут занять три строки.

Итак, вот перед вами два мира программирования. Почти все пришли к выводу, что мир управляемого кода значительно превосходит мир неуправляемого. Visual Basic был (а возможно, и остается) самым успешно продаваемым языковым продуктом всех времен, и разработчики отдавали ему предпочтение перед C или C++ в разработках для Windows, хотя слово «Basic» в названии продукта заставило более высокомерных программистов избегать его, несмотря на то, что это был довольно современный язык с элементами объектно-ориентированного программирования и весьма немногочисленными пережитками (номера строк и оператор LET ушли туда же, куда канула мода на обручи хула-хуп). Другой проблемой VB была необходимость поставлять для развертывания еще и библиотеку VB времени выполнения, что было очень неприятно для условно-бесплатного программного обеспечения, рассылаемого по модемным линиям, и, что еще хуже, обнаруживало перед другими программистами, что ваше приложение написано (позор!) на Visual Basic.

## *Единая библиотека времени выполнения для всех языков*

И вот появилась .NET. Это был грандиозный, потрясающий проект, призванный устранить всякую путаницу раз и навсегда. Разумеется, в нем должно быть управление памятью. В нем сохраняется Visual Basic, но обретается новый язык, по духу очень близкий Visual Basic, но с синтаксисом в стиле C, с фигурными скобками и точкой с запятой. И, что самое замечательное, новый гибрид Visual Basic/C будет называться Visual C#, поэтому больше не придется никому говорить, что вы программировали на «Basic». Все эти жуткие Windows-функции со своими концевиками, хуками, ошибками обратной совместимости и трудно постигаемой семантикой возврата строк выкидываются и заменяются единым ясным объектно-ориентиро-

ванным интерфейсом, в котором есть строки только одного вида. Единая библиотека времени выполнения для всех языков. Это было прекрасно. И формально они это осуществили. .NET – замечательная программная среда, которая управляет памятью и предоставляет богатый, полный и последовательный интерфейс к операционной системе и богатую, сверхполную и элегантную объектную библиотеку для базовых операций.

И тем не менее люди не очень активно используют .NET.

Нет, конечно, это относится не ко всем.

Но идея унифицировать программирование на Visual Basic и через Windows API путем создания *совершенно новой*, *созданной с нуля* программной среды, в которой будут не один, не два, а три языка (или их там четыре?), в некотором роде напоминает намерение прекратить ссору двух детей, крикнув еще громче, чем они; это срабатывает только на телевидении. В реальной жизни, если крикнуть «Замолчите!» двум людям, которые ссорятся, получается еще более громкая ссора – уже с тремя участниками.

(Между прочим, если кто-то из читателей следит за таинственным, но весьма политизированным миром форматов чтения синдицированных блогов, то он заметит, что в нем происходит то же самое. RSS раздроблен наличием нескольких версий, неточными спецификациями и политическими дискуссиями, а попытка навести порядок путем ввода *еще одного формата* под названием Atom привела к тому, что теперь есть несколько версий RSS плюс версия Atom, неточные спецификации и политические дискуссии. Пытаясь объединить две соперничающие силы с помощью альтернативной им, вы просто получаете три соперничающие силы. Ничего вы не объединили и ничего, по сути, не исправили.)

Так и .NET ничего не объединила и не упростила, а создала путаницу, в которой участвуют уже шесть языков, и все пытаются выяснить, какую же стратегию разработки принять, и можно ли переносить существующие приложения на .NET.

Несмотря на последовательную маркетинговую политику Microsoft под лозунгом «Переходите на .NET – доверьтесь нам!», большинство их клиентов по-прежнему использует C, C++, Visual Basic 6.0 и классические ASP, не говоря об остальных инструментах разработки от других компаний. А те, кто применяет .NET, разрабатывают с помощью ASP.NET веб-приложения, которые выполняются на сервере Windows, но *не требуют Windows-клиенты*, и это важный пункт, который я разберу подробнее, когда начну говорить о Вебе.

## *Подождите, это еще не все!*

Сейчас так много разработчиков отворачивается от Microsoft, что недостаточно изобрести заново весь Windows API, – надо изобрести его заново *дважды*. На прошлогодней конференции PDC они объявили о разработке новой основной версии своей операционной системы под кодовым названием Longhorn, в которой будет среди прочего совершенно новый API интерфейса пользователя, под кодовым названием Avalon, перестроенный с самого основания с целью воспользоваться преимуществами современных быстрых видеоадаптеров и выводом 3D в реальном времени. И если сегодня вы разрабатываете приложения с графическим интерфейсом для Windows с помощью WinForms – «официально» новейшей и лучшей среды программирования для Windows, то через два года вам придется начать все сначала, чтобы обеспечить поддержку Longhorn и Avalon. Отсюда ясно, что WinForms – мертворожденное дитя. Надеюсь, вы вложили в него не слишком много. Джон Уделл (Jon Udell) нашел слайд от Microsoft, на котором было крупными буквами написано «How Do I Pick Between Windows Forms and Avalon?» (Как сделать выбор между Windows Forms и Avalon?), и спрашивает: «А почему я должен выбирать между Windows Forms и Avalon?» Хороший вопрос, и удовлетворительного ответа на него он не нашел.[1]

Итак, у вас есть Windows API, есть VB, а теперь еще и .NET с несколькими языками, но не привязывайтесь слишком к какой-либо из этих сред, потому что, понимаете ли, мы делаем систему Avalon, которая сможет работать только на новейшей операционной системе Microsoft, которой еще *до-о-олго* ни у кого не будет. Лично у меня все еще не было достаточно времени глубоко изучить .NET, и мы не перенесли два приложения, выпускаемые Fog Creek, с классических ASP и Visual Basic 6.0 в .NET, потому что это не принесет нам прибыли. Никакой. Это все «суета и томление духа»;[2] насколько я понимаю: Microsoft мечтает, чтобы мы прекратили добавлять новые функции в наше ПО для отслеживания ошибок и в наше ПО для управления контентом, а потратили несколько месяцев на перенос их в другую программную среду, от чего не выиграет ни один клиент и что, следовательно, не принесет нам ни одного дополнительно проданного экземпля-

---

[1] John Udell «Questions about Longhorn, part 3: Avalon's enterprise mission», iDiscuss: Jon Udell's Weblog, InfoWorld, June 9, 2004. См. *weblog.infoworld.com/udell/2004/06/09.html#a1019*.

[2] См. главу 15.

ра, и что явится, следовательно, полной потерей нескольких месяцев, что хорошо для Microsoft, потому что у нее тоже есть ПО для управления контентом и ПО для слежения за ошибками, поэтому они ничего бы так не хотели, как чтобы я терял время, пытаясь догнать какую-то модную однодневку, а потом еще год или два потратил на изготовление версии для Avalon, пока они будут добавлять новые функции в свое собственное конкурирующее ПО. *Здо-о-о-рово*.

Ни у одного разработчика, которому есть чем заняться, не хватит времени, чтобы угнаться за всеми новейшими средствами разработки, исходящими из Редмонда, хотя бы потому, что *в Microsoft слишком много людей, которые делают инструменты для разработки*!

## *Это не 1990 год*

Microsoft выросла в период 1980-х и 1990-х годов, когда увеличение количества персональных компьютеров было таким поразительным, что каждый год продавалось больше компьютеров, чем их всего имелось до того. Это означало, что если сделать продукт, который работает только на новых компьютерах, то в течение одного-двух лет он мог захватить весь мир, даже если никто *не переходил* на него с других продуктов. Это было одной из причин, по которым Word и Excel полностью вытеснили WordPerfect и Lotus: Microsoft просто ждала очередной крупной волны обновлений аппаратной части и продавала Windows, Word и Excel корпорациям, осуществляющим очередной цикл закупки настольных компьютеров (иногда это был первый цикл). Поэтому во многих случаях Microsoft не приходилось трудиться, чтобы побудить своих клиентов перейти с продукта версии *n* на продукт версии *n+1*. Когда люди покупают новый компьютер, они с радостью устанавливают на него новейшее ПО от Microsoft, но значительно менее вероятно, что они станут обновлять его версии. Это было не так важно, когда производство PC расширялось подобно лесному пожару, но сейчас рынок насыщен компьютерами, многие из которых удовлетворяют своих пользователей, и в Microsoft внезапно начинают замечать, что требуется гораздо больше времени, чтобы распространить на них свои новейшие продукты. Они попытались проводить в последний путь Windows 98, но оказалось, что слишком многие продолжают ее использовать, и было обещано продлить поддержку этой скрипящей бабушки еще на несколько лет.[1]

[1] См. *www.windows-help.net/microsoft/98-lifecycle.html*.

К сожалению, попытки создать *новый* API и закрепить в нем пользователей при помощи таких отважных начинаний, как .NET, или Longhorn, или Avalon, оказываются не слишком успешными, если все продолжают работать на компьютерах выпуска 1998 г., которые еще достаточно хороши. Даже если Longhorn будет выпущена, как запланировано, в 2006 году, хотя я ни минуты не сомневаюсь, что этого не случится, пройдет еще пара лет, прежде чем она будет установлена на таком количестве компьютеров, что можно будет рассмотреть ее как платформу для разработки. Разработчики, разработчики, разработчики и разработчики не поддаются на шизофренические предложения страдающей раздвоением личности Microsoft относительно того, как они должны разрабатывать программное обеспечение.

## *И тут появляется Веб*

Странно, что мне удалось добраться то этого места и не упомянуть Интернет. Планируя создание нового программного приложения, каждый разработчик должен сделать выбор: можно делать его для Интернета, а можно делать «толстого клиента», который выполняется на PC. Основные аргументы «за» и «против» просты: веб-приложения проще развертывать, а толстые клиенты обеспечивают малое время реакции и гораздо более интересные интерфейсы пользователя.

Развертывать веб-приложения проще, потому что отсутствует процедура инсталляции. Установка веб-приложения заключается в том, чтобы ввести URL в окно адреса. Сегодня я установил новое почтовое приложение Google, введя Alt+D, gmail, Ctrl+Enter. Также значительно меньше проблем совместимости и совместного существования с другими приложениями. У всех пользователей вашего продукта оказывается одна и та же его версия, поэтому не приходится заботиться о поддержке букета прежних версий. Годится любая программная среда, потому что функционировать она должна только на сервере. Ваше приложение автоматически становится доступным практически для любого компьютера *на всей планете*. Данные ваших клиентов тоже автоматически становятся доступными практически на любом компьютере.

Но за это приходится расплачиваться гладкостью интерфейса пользователя. Вот несколько примеров того, что плохо получится в веб-приложении:

1. Создать быструю программу для рисования.
2. Организовать проверку орфографии в реальном времени с подчеркиванием красной волнистой линией.

3. Предупредить пользователей о том, что результаты их работы будут потеряны, если они щелкнут по кнопке завершения работы броузера.
4. Обновить малую часть экрана, модифицированную пользователем, без полного цикла обращения к серверу.
5. Создать быстрый управляемый клавишами интерфейс, не требующий мыши.
6. Позволить продолжить работу в отсутствие соединения с Интернетом.

Не все из этих проблем одинаково сложны. Некоторые вскоре будут решены остроумными разработчиками JavaScript. Два новых веб-приложения, Gmail[1] и Oddpost,[2] оба предназначенные для работы с электронной почтой, очень удачно находят обходные пути или полностью решают некоторые из этих проблем. А пользователей, похоже, не очень тревожат некоторые небольшие накладки интерфейса или замедленность реакции в Интернете. Почти все мои обычные знакомые почему-то вполне довольны веб-интерфейсом почты, несмотря на мои попытки убедить их, что «толстый клиент» богаче функциями.

Таким образом, переход на веб-интерфейс осуществлен уже процентов на 80, и даже до появления новых броузеров эта цифра может достичь 95%. Веб-интерфейса вполне хватает большинству людей, и это явно хорошая новость для разработчиков, которые продемонстрировали стремление разрабатывать почти любые существенные новые приложения в виде веб-приложений.

В результате вдруг обнаружилось, что API Microsoft не имеет такого большого значения. Веб-приложениям не нужна Windows.

Нельзя сказать, что в Microsoft не заметили происходящего. Конечно, заметили, и когда начали выясняться возможные последствия, они ударили по тормозам. Развитие многообещающих новых технологий типа HTA[3] и DHTML было остановлено. Бригады Internet Explorer не видно: она пропала без вести несколько лет назад. Microsoft явно не собирается допустить, чтобы DHTML стал еще лучше, чем он уже есть; это слишком опасно для их основного бизнеса, толстого клиента. В Microsoft сейчас *делают ставку на толстого клиента*. Таков лейтмотив каждой слайд-презента-

---

1 См. *gmail.google.com/*.

2 См. *www.oddpost.com/*.

3 Microsoft «Introduction to HTML Applications (HTAs)», Microsoft Corporation. См. *msdn.microsoft.com/workshop/author/hta/over view/htaover view.asp*.

ции о Longhorn. Джо Беда (Joe Beda) из бригады Avalon заявляет, что «Avalon и Longhorn в целом – это выражение твердой уверенности Microsoft в сохранении настольных компьютеров, локально выполняющих мощные приложения. Наши инвестиции направлены в настольные системы, мы считаем их вполне оправданными и надеемся поднять волну всеобщего энтузиазма...».[1]

По-видимому, слишком поздно.

## *Мне самому от этого немного грустно*

Мне действительно самому немного грустно от этого. Я считаю, что Интернет – это прекрасно, но веб-приложения с их скверными, медлительными и непривычными интерфейсами представляют собой большой шаг назад в юзабилити. Я люблю свои приложения толстого клиента и просто сойду с ума, если мне придется перейти на веб-версии приложений, которыми я пользуюсь ежедневно: Visual Studio, CityDesk, Outlook, Corel PHOTO-PAINT, QuickBooks. Но именно ими собираются снабжать нас разработчики. Никто (я снова имею в виду «меньше 10 000 000 человек») не хочет больше разрабатывать для Windows API. Венчурные капиталисты не станут вкладывать деньги в приложения Windows, потому что слишком боятся конкуренции со стороны Microsoft. А большинство пользователей не переживают (по крайней мере, в такой степени, как я) по поводу низкого качества веб-интерфейсов.

А вот убедительный показатель: я заметил (и мой знакомый агент по найму подтвердил это), что здесь, в Нью-Йорке, программисты Windows API, знающие C++ и программирование COM, зарабатывают около $130 000 в год, а обычные веб-программисты, пишущие на языках с управляемым кодом (Java, PHP, Perl и даже ASP.NET), – около $80 000 в год. Это огромная разница, и когда я обсудил это вопрос с некоторыми знакомыми в консалтинговой службе Microsoft, они признали, что Microsoft потеряла целое поколение разработчиков. Теперь необходимо $130 000, чтобы нанять программиста со знанием COM, и дело в том, что последние восемь или около того лет никто не стремился изучать программирование COM, поэтому приходится искать специалистов постарше, а они обычно уже занимают

[1] Beda, Joe (Джо Беда) «Is Avalon a way to take over the Web?» (Avalon – это способ завоевать Веб?), Channel 9, April 7, 2004. См. *http://channel9.msdn.com/ShowPost.aspx?PostID=948*.

административные должности, и уговаривать их пойти работать простыми программистами и заниматься (не приведи Бог) маршалингом и моникерами, моделью Apartment Threading и агрегированием и миллионами прочих вещей, которые, наверное, никто кроме Дона Бокса никогда не понимал, но даже Дон Бокс их больше видеть не может.

Как ни прискорбно, огромное количество разработчиков уже давно перешло на Веб и не собирается обратно. Большинство разработчиков .NET суть разработчики ASP.NET, создающие приложения для веб-сервера Microsoft. ASP.NET – великолепный продукт; я занимаюсь веб-разработками десять лет, и свидетельствую, что он просто на поколение опережает все остальные. Но это серверная технология, поэтому клиенты могут пользоваться любой настольной платформой. И он неплохо работает под Linux, в чем ему помогает Mono.[1]

Все это не сулит ничего хорошего ни Microsoft, ни доходам, которые она получала благодаря мощи своего API. Новым API оказывается HTML, а новыми победителями на рынке разработки приложений окажутся те, кто сможет заставить эту птичку петь.

[1] См. *www.go-mono.com/*.

ЧАСТЬ ЧЕТВЕРТАЯ

# Немного много о .NET

ГЛАВА СОРОК ТРЕТЬЯ

# Microsoft спятила

*Раньше Microsoft продавала программистам средства разработки. Я помню большую рекламу о Microsoft C, кажется, версии 3.0, в которой на четырех страницах занудно расписывались подробности новых методов оптимизации, примененных в компиляторе.*

*В какой-то момент, который мне трудно идентифицировать, продавцы продуктов Microsoft для разработчиков поняли, что по-настоящему крупными бюджетами распоряжаются не программисты, а исполнительные директоры. И этим директорам может понравиться болтовня типа «управление производительностью и масштабируемостью на протяжении всего жизненного цикла вашего приложения .NET для уменьшения рисков и снижения общей стоимости владения». (Это прямая цитата с домашней страницы Visual Studio.[1]) Прежние времена, когда продукты рекламировались для самих разработчиков, прошли давно. Впервые это пришло мне в голову, когда в июле 2000 года, примерно за три года до реального выпуска, Microsoft с большой помпой возвестила о .NET под аккомпанемент обычной рекламной болтовни, очевидно, для создания атмосферы страха, неуверенности и сомнений (Frustration, Uncertainty, Doubts – FUD) вокруг Java.*

22 ИЮЛЯ 2000 ГОДА

Последний анонс от Microsoft касается среды .NET, превозносимой журналами типа «Fortune» как огромная «революция», хотя в действительности

[1] Microsoft Visual Studio Development Center, *http://msdn.microsoft.com/vstudio* (по состоянию на 25 мая 2004 года).

это всего лишь туманные обещания (vaporware) и свидетельство, как мне кажется, очень серьезных проблем, возникших в Редмонде.

ПО туманных обещаний означает предложение всевозможных функций и продуктов, которые вам не продадут, потому что в действительности их не существует. Но .NET еще хуже. В своей пресыщенности и высокомерии Microsoft не потрудилась даже представить *сам туман*.

Почитайте внимательно доклад[1] и вы обнаружите, что несмотря на всю эту шумиху .NET представляет собой лишь тонкую пелену FUD. Там нет существа. Попробуйте разобраться с чем-нибудь, и окажется, что во всем докладе об этом *ничего не сказано*. Чем сильнее вы стараетесь, тем легче оно ускользает сквозь пальцы.

Я не хочу сказать, что в .NET нет ничего *нового*. Я хочу сказать, что там *нет ничего вообще*.

Вот цитата:

> Никто не сомневается, что Веб будет развиваться, но чтобы это развитие действительно открывало новые возможности для разработчиков, бизнеса и потребителей, необходимо радикально новое видение. Задача Microsoft в том, чтобы обеспечить это видение и технологию, которая превратит его в реальность.

А вот еще:

> Microsoft видит в .NET средство обретения новых возможностей потребителями, бизнесом, разработчиками ПО и индустрией в целом. Эта технология поможет полностью раскрыть потенциал Интернета. И Веб будет таким, каким вы хотите его видеть.

Что здесь происходит? Во всем документе я не нашел *ни единой идеи*, которую можно было бы действительно реализовать в программном продукте. Вместо списка функций Microsoft публикует список неопределенных «преимуществ» типа:

> Веб-сайты становятся гибкими службами, которые могут взаимодействовать, а также обмениваться и использовать данные друг друга.

---

[1] «Microsoft .NET: Realizing the Next Generation Interne.», Microsoft, 22 июня 2000 г. На сайте Microsoft этого документа больше нет, но сохранилась копия на *web.archive.org/web/20001027183304/http://www.microsoft.com/business/vision/netwhitepaper.asp*.

Вот она, «функциональность» этой замечательной архитектуры .NET. Из-за обширности, туманности и абстрактности она не значит *вообще ничего*, но это, похоже, никого не волнует. А как вам такая фраза:

> Microsoft .NET делает возможным поиск сервисов и людей для взаимодействия с ними.

Боже милосердный! Через *пять лет* после запуска Altavista и через два года после того, как Ларри Пэйдж и Сергей Брин изобрели радикально более эффективный механизм поиска (Google), Microsoft разыгрывает нас, утверждая, что средства поиска в Интернете отсутствуют и *она* решит для нас эту проблему. В таком же точно стиле написан весь документ.

Здесь надо отметить два обстоятельства. В Microsoft есть крупные мыслители. Размышляя, они обнаруживают закономерности. Они рассматривают проблему пересылки людьми друг другу файлов текстового процессора, а потом рассматривают проблему пересылки людьми друг другу электронных таблиц и открывают общую закономерность: люди посылают файлы. Так возникает один уровень абстракции. Затем они поднимаются еще на один уровень выше: люди *посылают* файлы, но веб-броузеры тоже «посылают» запросы для получения веб-страниц. Все это операции *посылки*, и наш умный мыслитель изобретает новую более высокую и широкую абстракцию, названную им *посылкой сообщений* (*messaging*), так что теперь все становится *совершенно* неясным, и никто уже не понимает, о чем они говорят.

Забираясь в подобном абстрагировании слишком высоко, вы лишаете себя кислорода. Временами эти мудрые мыслители не могут вовремя остановиться и создают нелепые универсальные абстрактные картины вселенной, которые замечательны, но не имеют абсолютно никакого смысла.

Похоже, такой же случай произошел и здесь:

> Платформа .NET представляет собой новое поколение настольных платформ Windows и поддерживает высокую эффективность, творческую активность, менеджмент, отдых и многое другое, будучи нацеленной на установление контроля пользователей за их цифровой жизнью.

Это настолько абстрактно, что не поддается критике. Кто же не хочет операционной системы, которая поддерживает высокую эффективность? Замечательная функция! Дайте мне какую-нибудь классную новую операционную систему с высокой эффективностью! Непонятно только, как именно собирается Microsoft ее создавать? За последние 20 лет эффективность ПО росла медленно и постепенно. Они что – открыли какое-то но-

вое химическое вещество, которое позволит им сделать свою операционную систему более эффективной? Я в этом сомневаюсь. Я думаю, что они блефуют. FUD и химеры.

## *Страшно то, что они говорят об этом серьезно*

Я знаю Microsoft – я там три года проработал. И я знаю тот тип людей, которые участвовали в написании этого документа. Билл Гейтс почти наверняка сыграл в этом большую роль; он для того и ушел с поста президента, чтобы заниматься такими вещами. Я не думаю, что в Microsoft создали этот документ, потому что им нужно химерное программное обеспечение. Они очень умные люди.

Я думаю, что они на самом деле убеждены в том, что создают будущее и знают, как это делать. Они рассмотрели все продукты Microsoft, от Hotmail до SQL Server, и постарались вписать их в «концепцию смелого нового видения». Но беда в том, что на самом деле никто там не изобрел ничего эпохального. И это неудивительно – не потому, что в Microsoft собрались дураки, это не так, а потому, что эпохальные изобретения очень редки, а количество умных людей в Microsoft ограничено. Только один человек на всем свете изобрел Napster, и он не работал в Microsoft. Microsoft отчаянно старается уверить себя, что может совершить революцию, но даже во время кембрийского взрыва Интернета ежегодно возникает лишь горстка подлинно революционных идей, и шансы, что одна из них возникнет в крохотном мире Билла Гейтса и рыцарей редмондского стола, ничтожно малы. Шансы окажутся еще меньше, если подумать, что когда у типичного умного программиста, трудящегося где-то в недрах Microsoft над драйвером дисплея для Windows NT, возникнет замечательная идея, к ней, скорее всего, никто не прислушается.

Единственное конкретное, что можно уловить из этого документа, это то, что программное обеспечение должно быть платной службой Интернета, а не устанавливаться с CD-ROM.

Для пользователя получение текстового процессора по подписке из Интернета, а не с CD-ROM, может иметь некоторую выгоду, хотя... нет, едва ли. На практике это не решает никаких проблем пользователя. Получать исправления ошибок через Интернет? Замечательно. Я уже могу это делать. Я загружаю патчи для продуктов Microsoft в течение семи лет, и сейчас это происходит вполне автоматически. Получать новые версии? В чем смысл, если единственное, что отличает новую версию, это возможность

проще загружать новые версии! За последние три выпуска Word в него едва ли добавилась хоть одна новая функция, за исключением того, что в какой-то момент они сделали нечто непонятное для «облегчения» позиционирования картинок, и теперь они никогда не попадают туда, куда мне нужно.

Правда заключается в том, что еще в 1991 году в Microsoft заметили, что все большая часть их доходов поступает от обновления версий и что трудно заставить всех переходить на новые версии, и их попытки заставить своих клиентов перейти на модель приобретения программ по подписке длятся почти десятилетие. Но они не были успешными, потому что клиентам это не нужно. Microsoft видит в .NET способ все-таки вынудить клиентов принять модель подписки, которая соответствует ее конечным целям.

Очень похоже, что Microsoft .NET, не удовлетворяя ни единой потребности клиентов, удовлетворяет потребность Microsoft занять чем-то 10 000 программистов на ближайшие десять лет. Всем известно, что они уже давно не придумывали никакой функции текстовой обработки, которая была бы нужна всем, так чем же еще собираются заниматься все эти программисты?

## *Светлая сторона «видения»*

Есть старый анекдот. Человек приходит к психиатру. Тот показывает ему изображение птицы и спрашивает: «О чем вы подумали, глядя на эту картинку?». Человек говорит: «О сексе». Психиатр показывает ему изображение дерева и спрашивает: «А о чем вы подумали, глядя на *эту* картинку?» Человек говорит: «О сексе». Картинка поезда. «О сексе». Картинка дома. «О сексе».

«Боже мой! – говорит психиатр. – Да вы сексуальный маньяк!»

«Кто маньяк – *я*?! – говорит человек. – А *вы* зачем все время показываете мне эти грязные картинки?»

Положительная черта таких туманных документов, как доклад о .NET, в том, что они играют роль своего рода теста Роршаха. Люди, имеющие какие-то свои предварительные идеи, читают этот документ, и поскольку он составлен так туманно, им начинает казаться, что Microsoft повторяет их идеи. У Дэйва Винера (Dave Winer),[1] президента UserLand Software,[2] есть много интересных новаторских идей относительно программного обес-

[1] См. *wwwscripting.com/*.

[2] См. *wwwuserland.com/*.

печения. Когда он прочел о .NET, то решил, что Microsoft наконец-то признала те идеи, о которых он говорил уже два года. Нет, Дэйв, слишком много чести для них. По сравнению с вами они абсолютно бестолковы. Они пытаются прибегнуть к тому же фокусу, что и горячие линии психиатрической помощи и газетные гороскопы: посредством туманных и бессмысленных обобщений они заставляют вас поверить, что могут читать ваши мысли. «Сегодня планеты расположились так, что вам предстоит сделать большой шаг в достижении ваших целей.» Разница в том, что у Дэйва есть реальные конкретные идеи, которые можно перевести в реальное программное обеспечение, в то время как Microsoft по-прежнему занимается такой же болтовней, как и шесть лет назад, когда они рассказывали, как «Cairo» будет предоставлять информацию «на кончиках ваших пальцев» – видение, которое воплотил Интернет, но не Cairo.

Поэтому вся эта бессмысленная болтовня в конечном счете приведет в движение чьи-то творческие силы (как в UserLand) и породит какие-то реальные нововведения. Но эти нововведения, скорее всего, придут не изнутри Microsoft, а снаружи.

*Постскриптум: После появления этой статьи почти все продукты Microsoft получили новые названия с добавлением «.NET»... на некоторое время, пока путаница не стала совершенно невообразимой. С большим трудом и затратами Windows Server 2003 был переименован из «.NET Server», и область действия марки .NET была ограничена новой программной средой «управляемого кода». Которая, кстати, действительно хороша. Сочетание C# с общей языковой структурой .NET действительно образует фантастическую программную среду. Она даже уменьшает риск и снижает общую стоимость владения! Нирвана!*

ГЛАВА СОРОК ЧЕТВЕРТАЯ

# Наша стратегия .NET

11 апреля 2002 года, четверг

Вот как я себе представляю сейчас постепенный переход на средства разработки .NET в Fog Creek.

Текущее состояние: CityDesk написан в основном на Visual Basic 6.0, а некоторые его части – на Visual C++ 6.0. FogBUGZ написан в основном на VBScript для ASP, но некоторые его части – на C++. Почти все инструменты для внутреннего применения и наше присутствие в Веб (FogShop, Discussions и т. д.) написаны на VBScript для ASP.

Зачем вообще переходить на .NET? Если коротко, то потому, что .NET сейчас представляет собой одну из самых блестящих и эффективных сред для разработки из всех, когда-либо созданных. ASP.NET действительно позволяет очень легко создавать полезные веб-приложения. Пару последних дней я с потрясающей скоростью строю некоторые приложения, которые мы используем внутри фирмы. Всякая дрянь, отнимающая три четверти времени при создании веб-приложений с помощью ASP (например, проверка данных формы и сообщение об ошибках), реализуется тривиальным образом. ASP.NET дает такой же скачок продуктивности по сравнению с ASP, как Java по сравнению с C. Черт побери.

В C# вошла большая часть удачных решений Java, с небольшими усовершенствованиями типа автоматического боксинга. Хотя мы и раньше делали все, что могли, для разумной объектной ориентированности кода в ASP и VB6, перейти на C# будет полезно.

Наконец, с .NET поставляются замечательные библиотеки классов. Было переработано *все* – от доступа к данным и веб-разработки до создания

GUI, поэтому создалось редкостное единообразие, сверху до низу. Если взглянуть на старые API Win32, то поражает количество способов, которыми, например, можно получить обратно строку при вызове функции. Каждую пару лет менялось представление о том, как лучше всего это сделать. Порядок наступил с появлением .NET. Мне очень нравится, что можно воспользоваться графическим элементом календаря в ASP.NET, который генерирует HTML, позволяющий выбрать дату, и быть уверенным, что класс «даты», который при этом возвращается (кажется, `System.DateTime`), будет в точности тем классом даты, который нужен классам SQL Server. Вы не поверите, как много времени мы тратили в прошлом на такие вещи, как переформатирование дат для команд SQL или преобразование `COleDateTimes` в `НекоторыйДругойТип_DateTime`. И наконец, строковый тип данных, который применим всюду! Лишь на прошлой неделе я писал код ATL и возился со всеми этими `BSTR`, и `OLECHAR`, и `char*`, и `LPSTR`, и это была ужасная путаница. Слава Богу, избавились.

Да, я согласен: .NET нарушает закон «никогда не переписывай с чистого листа».[1] Microsoft это сошло с рук по двум причинам. Во-первых, у них был лучший в мире конструктор языков, человек, которому мы обязаны 90% прироста эффективности разработки программ за последние 20 лет, Андерс Гейльзберг (Anders Hejlsberg),[2] давший нам Turbo Pascal (спасибо!), Delphi (спасибо!), WFC (отлично!) и теперь .NET (сногсшибательно). Во-вторых, они посадили за эту работу тьму инженеров на целых три года, в какой-то мере ослабив на это время свое участие в конкурентной борьбе. Запомните: если Microsoft может что-то себе позволить, это не значит, что то же самое можете вы. *Microsoft создает свое гравитационное поле*. Обычные правила к ним неприменимы.

Я, наверное, получу десятки сердитых писем, авторы которых станут расхваливать какую-то другую среду разработки либо начнут спрашивать, почему мы просто не используем Java и возможность «написать один раз и выполнять везде» (ха-ха), или Delphi (талант покинул ее, .NET – *это и есть* Delphi 7.0, 8.0 и 9.0), или Lisp, или что-то там еще. «Меня запирают в сундук Microsoft!» – скажут они. К сожалению, у меня нет сейчас времени вступать в религиозные дискуссии, и они, как правило, мне неинтересны.

---

[1] См. главу 24.

[2] Osborn, John (Джон Осборн) «Deep Inside C#: An Interview with Microsoft Chief Architect Anders Hejlsberg» (Глубины C#: интервью с главным архитектором Microsoft Андерсом Хейлсбергом), O'Reilly.com, August 1, 2000. См. *windows.oreilly.com/news/hejlsberg_0800.html, http://www.gotdotnet.ru/LearnDotNet/CSharp/731.aspx*.

Даже если японский язык лучше английского, меня это *не волнует*. Это не имеет значения. Позвольте мне закончить описание нашей стратегии.

Первая проблема: мы недостаточно хорошо знаем .NET, чтобы писать хороший код. Как и любая среда разработки, она предоставляет много путей решения каждой задачи, и мы еще не вполне освоили первый путь, не говоря уже о втором. Поэтому качество кода .NET, который мы можем писать, недостаточно для готового продукта. Пока не вышла *первая* книга Билла Вона (Bill Vaughn) по ADO, мы даже не знали оптимальных способов выполнения простых запросов SQL. Поэтому наша первая задача – это обучение, которое примет форму выполнения последующих разработок «для себя» и для Интернета на .NET. Практически это все те программы, за которые нам не платят денег. Мы можем частично перенести на .NET FogShop и, конечно, использовать .NET для всевозможных внутренних работ. (Сегодня я написал на ASP.NET генератор купонов для FogShop. Не очень изящно, но работает!)

Вторая проблема: эта толстенная 20-мегабайтная CLR (библиотека времени исполнения). И так достаточно скверно, что из 8 мегабайт, которые надо загрузить для установки CityDesk, 6 приходится на библиотеки времени исполнения и доступа к данным; трудно рассчитывать, что каждый домашний пользователь CityDesk Starter Edition захочет дополнительно загружать 20 Мбайт. Надеемся на то, что через год-два у многих будет установлена CLR, полученная из других источников (очень жаль, что ее нет в составе Windows XP). Мы будем за этим следить.

Короче говоря, ни CityDesk, ни FogBUGZ сегодня переносить на .NET нельзя. Мы перенесем какую-нибудь из новых версий CityDesk, когда CLR достигнет 75-процентного распространения. План таков:

1. Портировать имеющийся код и формы с помощью инструментов Microsoft для конвертирования.
2. Исправлять код, пока он не заработает снова.
3. Создавать новые формы и классы с помощью C#.
4. Постепенно переносить старые формы и классы на C# – во всяком случае те, которые потребуют значительных изменений.
5. Оставить многочисленные старые формы и классы в VB.NET (с помощью уродливых строковых функций обратной совместимости и т. п.), если они нормально функционируют.

FogBUGZ тоже придется подождать, пока CLR не распространится шире на компьютерах-серверах; необходимо провести изучение наших клиен-

тов и выяснить, насколько они будут недовольны, если FogBUGZ потребует CLR.

У нас в работе есть еще один продукт, о котором мы не говорили во всеуслышание. У него будет значительная часть кода, общая с FogBUGZ (подмножество, которое мы назовем «Dispatcho»), поэтому он останется в основе VBScript/ASP, пока мы не перенесем FogBUGZ.

Для FogBUGZ/Dispatcho/Секретный Новый Продукт план таков:

1. Подождать, пока можно будет безбоязненно требовать CLR.
2. Перенести существующие классы «бизнес-логики» на C#.
3. Оставить текущие веб-формы в ASP.
4. Создавать новые веб-формы в ASP.NET.

ГЛАВА СОРОК ПЯТАЯ

# Простите, сэр, можно мне взять компоновщик?

*Будь я параноиком, то решил бы, что Microsoft на самом деле не хочет создавать инструментальные средства для разработчиков систем или приложений, которые могли бы конкурировать с ее собственным главным бизнесом. На самом деле они хотят делать лишь инструментальные средства для отделов ИТ, в которых пишут заказное вертикальное ПО и где нет перспектив для Microsoft. Но это уже паранойя.*

28 января 2004 года, среда

По какой-то причине в великолепной и новейшей среде разработки Microsoft для .NET отсутствует один важнейший инструмент... инструмент, который стандартно включается в среду разработки, наверное, с 1950-х годов, и его наличие подразумевается как само собой разумеющееся. Поэтому очень странно, что никто не заметил его отсутствия в .NET.

О чем идет речь? О *компоновщике*. Вот в чем состоит его функция. Он объединяет откомпилированный вариант программы с откомпилированными версиями всех библиотечных функций, которые задействованы в программе. Затем он удаляет все функции, которые в программе не используются. Наконец, он создает единый исполняемый двоичный код, способный работать на компьютерах пользователей.

Вместо этого в .NET эксплуатируется идея «библиотеки времени выполнения» – 22-мегабайтной дымящейся кучи кода, которая должна быть на компьютере у каждого, кто захочет выполнять приложения .NET.

Библиотеки времени выполнения составляют проблему, аналогичную DLL, потому что могут возникнуть проблемы, если приложение версии 1 было разработано для библиотеки времени выполнения версии 1, а потом появляется библиотека времени выполнения версии 2, и приложение версии 1 вдруг по непонятным причинам перестанет работать. Например, в данное время по какой-то причине наша внутренняя система учета стала округлять объем продаж до 4 десятичных знаков, когда мы перешли с версии библиотеки времени выполнения 1.0 на версию 1.1. Обычно несовместимости носят более неприятный характер.

На самом деле в .NET есть развитая технологическая система, называемая «декларациями» (manifests), которые весьма сложны и должны гарантировать использование каждым приложением *только* необходимой ему версии библиотеки времени выполнения, но я не знаю никого, кто разобрался бы, как с ними работать.

Расскажу одну историю. Мы хотели, чтобы на новогодней вечеринке в Fog Creek группа компьютеров в главном помещении вела обратный отсчет времени, оставшегося до полуночи. Майкл написал для этого приложение на C# с использованием WinForms примерно за 60 секунд. Это замечательная среда разработки.

Моей задачей было запустить countdown.exe на трех компьютерах. Казалось бы, легко.

Как бы не так. Двойной щелчок по EXE, и я получаю возмутительно недоброжелательное сообщение об ошибке, вызванной отсутствием mscoree.dll или чего-то еще такого, за которым любезно следует дамп моего маршрута поиска. Никакого намека на то, что проблема была лишь в том, что не установлена библиотека времени выполнения .NET. Поскольку я программист, то понял, что проблема должна быть в этом.

Как установить библиотеку времени исполнения? «Простейший» способ – через функцию обновления системы Windows Update. Но Windows Update потребовала, чтобы я установил все критически важные обновления *перед* тем, как устанавливать библиотеку времени выполнения. Разумно, не правда ли? Двумя «критически важными» обновлениями оказались сервиспак Windows и новая версия Internet Explorer, и после установки каждого из них потребовалась перезагрузка.

Короче, для каждого компьютера, на котором мне надо было выполнить это маленькое приложение .NET, я был вынужден загрузить 70 или 80 Мбайт (благо у нас скоростное соединение с Интернетом) и перезагрузиться три или четыре раза. И это в софтверной компании! Я знаю,

сколько времени это заняло, потому что, начав первую загрузку, я запустил «Office Space» на широкоформатном телевизоре, и к тому моменту, когда кончился фильм, процедура инсталляции была *почти* завершена. Пока шел фильм, я каждые десять минут должен был вскакивать, подходить к каждому компьютеру и нажимать «OK» в каком-нибудь дурацком диалоговом окне.

Такая процедура достаточно обременительна даже для наших внутренних приложений. Но я думаю о нашем продукте CityDesk. Почти все наши пользователи загружают бесплатную пробную версию, прежде чем купить продукт.[1] Загружаемый модуль имеет размер около 9 Мбайт и не предъявляет никаких дополнительных требований. Почти ни у кого из наших пользователей пока нет библиотеки времени выполнения .NET.

Если бы наши предполагаемые пользователи, а это, как правило, небольшие организации и домашние пользователи, для одного лишь ознакомления с нашим продуктом должны были выполнить инсталляцию продолжительностью в целый фильм, то, наверное, мы потеряли бы 95% из них. И это еще не клиенты, а кандидаты в них, и я не могу позволить себе потерять 95% от их числа ради того, чтобы иметь более удобную среду разработки.

«Но, Джоэл, – скажут мне, – в конце концов библиотека времени выполнения появится у многих, и эта проблема будет снята».

Я тоже так считал, но потом обратил внимание, что каждые шесть или двенадцать месяцев Microsoft поставляет *новую версию* библиотеки времени выполнения, и постепенно растущее количество людей, у которых установлена новая версия, снова падает до нуля. И будь я проклят, если стану стремиться протестировать свое приложение с тремя разными версиями библиотеки времени выполнения только для того, чтобы удовлетворить те 1,2% всех клиентов, у которых есть эта одна из трех.

Мне просто надо скомпоновать все, что необходимо, в одном статическом EXE-модуле, который будет выполняться без всяких дополнительных инсталляций. Пусть он будет немного больше. Все, что мне потребовалось бы, – это функции, которыми я реально *пользуюсь*, интерпретатор байткода и небольшая библиотека времени выполнения. Мне не надо, чтобы в нее входил весь компилятор C#. Я вас уверяю, что в CityDesk не требуется компилировать никакого исходного кода C#. Мне не нужны все 22 мегабайта. Скорее всего мне нужны 5 или 6 Мбайт, *не больше*.

---

[1] См. дополнительные сведения на *www.fogcreek.com/CityDesk/Starter.html*.

Я знаю компании, у которых есть технологии, позволяющие это сделать, но они не могут пользоваться ими без разрешения Microsoft распространять отдельные части библиотеки времени исполнения, например интерпретатор байт-кода. Поэтому я обращаюсь к Microsoft: «Проснись и дай нам немножко замечательной технологии компоновки 1950-х годов и позволь мне сделать единый EXE, который выполняется на любом компьютере с Windows 98 или более новой системой *без всяких внешних зависимостей*.» В противном случае .NET окажется фатально дефектной средой для клиентского, загружаемого программного обеспечения.

ЧАСТЬ ПЯТАЯ

# Приложение

ПРИЛОЖЕНИЕ

# Лучшие вопросы и ответы с Ask Joel

*Некоторое время я вел на своем сайте дискуссионный форум «Ask Joel» (Спроси у Джоэла), на котором предлагал читателям задавать вопросы и пытался отвечать на них. Вот некоторые из моих любимых вопросов.*

**В** За последние годы мою организацию заполонили менеджеры программ, которые лучше разбираются в футболках и штанах, чем в технических вопросах.

Они усиливают общую неразбериху, неправильно употребляя взаимонезаменяемые технические термины. Они не видят технических тонкостей. Хуже того, они не в состоянии противодействовать влиянию плохих инженеров. (А может быть, действительную проблему у нас составляют слабые инженеры – может быть, софтверным компаниям не нужны технически грамотные менеджеры программ?)

Может быть, я злобный зануда, чьи оценки несколько искажены личными разочарованиями и тоской по тем дням, когда маленькое язвительное письмо было способно сразу похоронить дурацкую идею. Но какие можно принять более сдержанные меры? Насколько глубоко должны быть технически подготовлены МП по сравнению с теми, с кем они работают? Или я здесь совершенно неправ?

*– Анонимный автор*

**О** Эх, футболки и «хаки»... Как я скучаю по Редмонду. Где еще можно пойти в воскресенье в Gap, придти на следующий день на работу и обнаружить, что еще двенадцать менеджеров программ надели на себя абсолютно то же, что ты вчера купил?

Мне уже приходилось говорить, что менеджеры программ, которые относятся без уважения к разработчикам, не будут эффективны, потому что не смогут ничего довести до конца. Во времена моего участия в команде Excel разработчики могли съесть технически неподготовленных менеджеров программ со всеми потрохами. Судя по вашим рассказам, вам попадались бригады, сочетавшие в себе худшее: слабых менеджеров программ со слабыми разработчиками.

Ген управления программой и ген разработки программ несовместимы между собой, и это факт. Идеальный менеджер программы – это разработчик программного обеспечения, обладающий сочувствием к пользователям, организационными навыками и человеческими качествами лидера женского сообщества, и таких просто не найти в достаточном количестве. В какой-то момент кто-то решил, что умение писать код не нужно для этой работы, и что получилось? Кто проводит собеседования с кандидатами в менеджеры программ в Microsoft? Другие менеджеры программ.

Единственная причина, по которой мне удавалось что-то как руководителю программы, – это наличие у меня двух – да, двух – полных комплектов ДНК. Это называется счетверенной спиралью, и такая есть только у меня. И я полностью избегал Gap; после одного ужасного случая («о боже, это Я САМ прошел только что мимо себя в холле?») и многочисленных столкновений свитеров с одинаково смелой полосатой раскраской я стал покупать спортивные рубашки и брюки в других 27 совершенно нормальных магазинах спортивных рубашек в торговом центре Бельвю-Сквер.

Поговорите с менеджерами Microsoft о том, как они представляют себе свою роль. Они обязательно споют вам песню про то, что «их задача не бегать с мячом по полю, а прежде всего найти этот мяч!» или нечто в этом роде. Итак. Программисты считают, что они решают все, и менеджеры программ считают, что они решают все. Разве могут и те и другие решать все? Не могут. Кто же тогда решает на самом деле? Попробую подсказать. Из тех менеджеров программ и разработчиков, с которыми ты знаком, кто, в целом, лучше умеет общаться с людьми? А? Громче, не слышу. Вот! Конечно, менеджеры программ. И ты это знал. Это разработчики не могли использовать в своих интересах умение общаться на приеме для летних стажеров на вилле Билла Г. у озера. Разработчики так плохо умеют общаться с людьми, что они даже не могут представить себе, для чего нужно уметь общаться с людьми, кроме как чисто гипотетического назначения гипотетического свидания («мне... нравятся... большие *задницы*, но я не умею *врать*...»),

поэтому неудивительно, что они даже не подозревают о существовании того секрета, который я могу сегодня открыть.

Вы не замечали, что действительно хорошие менеджеры программ заставляют вас поверить, будто вы делаете всю важную работу проектирования, а они – лишь мальчики на побегушках, занимающиеся такой тупой бюрократией, как «отношения с пользователями», «пачкотня с маркетингом» и «составление дурацких спецификаций»?

Самое необходимое качество менеджера программы – это научиться заставлять программистов делать то, что вам нужно, путем убеждения их в том, что это была *их собственная идея*. Да, чтобы добиться этого, необходима футболка, но в среднем менеджер программы занимается этим три или четыре раза в день и по большей части добивается своего. Я читал целые самоизданные «книги» бывших разработчиков из Microsoft, которые нестерпимо скучно и подробно рассказывали о своем самодовольном поведении в Microsoft и в том числе отводили целую главу описанию тупости и непригодности менеджеров программ. При этом им ни разу не пришло в голову, что ими постоянно манипулировали, – признак блестящей работы менеджера программы. Блестящий менеджер программы заставляет поверить, что все это происходит без специальных усилий! И это нетрудно, когда имеешь дело с людьми, любимая реплика которых: «Это не мой ответ плох, плох ваш вопрос!»

Можете представить себе, какое требуется мастерство, чтобы склонить разработчика сделать то, что вам нужно, и одновременно выглядеть дураком и массовиком-затейником? Нужно не только позабыть про свое эго («Ах, какая замечательная идея, м-р Разработчик!») и при этом выглядеть, как занятый самим собой студент. Шон Пенн мог бы сыграть такую роль так же успешно, как это делает средний менеджер программы в Microsoft.

**В Вот сейчас «горячая новая штучка» – это .NET. И надвигается Longhorn. Эх. Но как вам кажется, к чему придет программный бизнес, скажем, через 10 лет? Как изменится профессия разработчика за это время? Какие новые концепции придется освоить? Где могут произойти решающие прорывы?**

*– Норрик*

О Посуду на кухне станут мыть роботы! А в лифтах роботы станут нажимать на кнопки!

Вместо кассиров в банках будут стоять ряды сверхсовременных компьютеров, выдающих пачки новых хрустящих долларов. (В будущем даже обычные рабочие станут зарабатывать столько, что, приходя в банк, смогут снимать по десять, двадцать, а то и сорок долларов!)

Летательные аппараты станут переносить нас в удобных герметичных кабинах из одного города в другой. Возможно, там, высоко над облаками, нам даже будут подавать какую-нибудь простую горячую еду!

К самой большой в мире энциклопедии – больше школьной библиотеки – можно будет мгновенно получить доступ в каждом доме с помощью воздуховодов или электрических проводов.

Специальные телефотографические аппараты смогут распространить образование по всему свету, давая даже обычному мусорщику возможность самосовершенствоваться, изучая классику, ликвидируя голод и страдания.

Да, и еще, в C# появятся анонимные методы, позволяющие записывать связанный с делегатом код в том месте, где используется делегат, удобно прикрепляя этот код к экземпляру делегата.

**В Моя компания скоро закончит разработку нашего первого продукта. Наша идея не слишком оригинальна, но в нашей нише есть еще всего один или два игрока, и мы считаем, что наш продукт может быть лучше, чем у них. Мы добавили пару дополнительных функций, которых нет у конкурентов и которые, как нам кажется, имеют некоторую ценность. Однако мы только что обнаружили, что наш главный конкурент собирается выпустить новую версию своего продукта с новыми функциями, которые очень похожи на наши.**

**Понятно, что мы пришли в некоторое уныние. Мы считаем, что наш продукт очень многое делает лучше, но он очень похож на уже существующие, у которых было больше времени, чтобы достичь зрелости.**

**Какими должны быть наши главные цели в таких условиях? Найти новые функции, которые дали бы нам явное преимущество в конкуренции? Работать над совершенствованием нашего продукта и набора имеющихся в нем функций, чтобы вывести нас вперед? Буду благодарен за любой совет.**

***– Сэм Томас***

О Перестаньте думать о своих конкурентах. Повторяйте за мной: прислушивайтесь к своим клиентам, а не к своим конкурентам!

Прислушивайтесь к своим клиентам, а не к своим конкурентам!

Прислушивайтесь к своим клиентам, а не к своим конкурентам!

У нашего пакета для учета ошибок есть десятки конкурентов, и я понятия не имею, что они делают, и лучше они или хуже, чем наш. Мне это совершенно безразлично. Меня волнует только то, что говорят мне мои клиенты, и их мнение дает мне более чем достаточно материала для работы.

Выпускайте свой продукт и не обращайте внимания на конкурентов. У вас появится некоторое количество клиентов, которые *понятия не имеют*, что существуют конкурирующие продукты. Возможно, их будет немного, но это начало. Регулярно обращайтесь к своим клиентам и следите, чтобы на каждой странице вашего сайта был почтовый адрес. Вы станете замечать тенденции – неоднократное упоминание функций вашего продукта, которые не устраивают людей. В FogBUGZ 1.0 такой функцией была возможность прикрепить файл к записи об ошибке. Мы включили ее в FogBUGZ 2.0. Есть ли такая функция у наших конкурентов? Убей меня Бог, не знаю. Мне жаль тех десяти минут, которые нужны, чтобы выяснить это. Меня волновало только одно: люди сообщали, что не купят наш продукт, пока нельзя будет прикреплять к ошибкам файлы. Теперь это можно делать.

В «Третьем письме о стратегии» есть ряд мыслей о том, как заставить пользователей перейти на ваш продукт с продукта сильного конкурента.[1] Но сейчас вы должны выпустить продукт, а затем послушать своих клиентов и тех, кто почти готов ими стать, чтобы выяснить направление дальнейших действий.

## В продолжение...

**Какими методами вы побуждаете своих клиентов предоставлять вам содержательную информацию? Как насчет предполагаемых клиентов? Занимаетесь ли вы обзвоном/рассылкой писем/чем-то еще с этой целью?**

***– dir***

О Никаких звонков, писем и т. д. Вот три главных для нас средства получения обратной информации от клиентов:

- Пункт меню «Send Feedback» в CityDesk. Письмо сразу попадает в нашу базу данных учета ошибок и создает такой большой поток (отличной) обратной информации, что мы не в состоянии отвечать на

[1] См. главу 38.

письма; в лучшем случае мы успеваем подсчитать голоса в пользу тех или иных предложенных для реализации функций.

- Общая политика помещения ссылки на электронную почту на каждую страницу нашего сайта.
- Дискуссионные форумы в Интернете.

Эти три метода дают нам более чем достаточно обратной информации. Обычно я прошу Дмитрия (который осуществляет у нас основную часть технической поддержки) следить за такими сообщениями, как «мне нравится ваш продукт, но я не могу купить его из-за X».

Такие высказывания гораздо важнее, чем вопросы типа «почему в вашем продукте нет X?», потому что последние могут и не выражать «окончательного отказа», а могут оказаться простой риторикой или проявлением потребности в творчестве. Например, кто-то прочел одну из моих статей по ссылке на Slashdot и хочет теперь знать, работает ли CityDesk под Linux; это совсем не значит, что человек уже сидит с кредитной карточкой наготове. Или кто-нибудь попользовался продуктом, и у него возникла «замечательная мысль» по поводу некой новой функции, что отрадно слышать, но что не так важно, как сообщение «мне нравится ваш продукт, но я не могу купить его из-за X».

**В Меня озадачила одна простая вещь, касающаяся снабжения. Почему во всех продуктовых магазинах упаковывают покупки в двойные пакеты? Почему бы просто не сделать пакеты покрепче?**

**Даже в Duane Reade, где для больших предметов есть более прочные пакеты, мне упаковали бутылку воды в двойной пакет. Или я не понимаю чего-то очевидного?**

***– Любопытный из Н.-Й.***

О Это потому, что кассиров не спрашивают, какими должны быть мешки. Бухгалтеры международной штаб-квартиры Gristede's живут в дальнем Квинсе и в супермаркеты ездят на SUV, и им просто неведома проблема, как донести продукты до дома. Они просто закупают то, что дешевле всего за штуку, даже если в конечном итоге это оказывается дороже, потому что они не были на Манхеттене со времен большой аварии электроснабжения и бунтов середины 1970-х, исключая одну поездку на дневное представление «О, Калькутта» перед ее закрытием в 1980-х, и в самых кошмарных снах они не могут представить себе, что это такое – тащить 12 бу-

тылок Margarita Mix целых три квартала по городу, а потом еще шесть пролетов вверх по лестнице в студию за 1700 в месяц в «Адской кухне».

**В Сколько времени (в процентах) надо отвести в качестве резерва при оценке времени реализации проекта, и что следует при этом учитывать?**

***– Вэни***

О Если ваши оценки основываются на очень детализированных задачах (все задачи – в пределах одного дня) и вы достаточно опытны, чтобы включить в оценки *все* (в том числе ежегодные отпуска, праздники, отпуска по болезни, время, необходимое для интеграции, время для «новых функций», придуманных в процессе разработки, время на дурацкие административные совещания, время на собеседования с поступающими на работу и т. д.), то достаточно оставить 10-процентный резерв.

Вместо того чтобы выделять резерв и просто отмахиваться от проблем, следует различать разные типы резервов времени и распределять для них время в соответствии с приоритетами:

- Резерв для новых функций, придуманных в процессе разработки.
- Резерв для неожиданных задач, возникших из-за действий конкурентов.
- Резерв для интеграции фрагментов кода, написанных разными разработчиками, чтобы обеспечить их совместную работу. (В зависимости от опыта бригады он может составить от 25 до 100%.)
- Резерв для поиска и исправления ошибок, найденных при тестировании.
- Резерв для заданий, не связанных с разработкой, которые должны выполняться служащими, например, «обязательные однодневные занятия по повышению взаимозаменяемости», «внеочередные совещания служащих компании», пожарные тренировки, торт и поздравление босса с днем рождения и т. д. и т. п.
- Резерв для тех задач, которые заняли больше времени, чем предполагалось.
- Резерв для тех задач, продолжительность которых не удалось заранее оценить.

Разбейте свой резерв подобным образом, и вы сможете тщательнее за ним следить. Если вы потратили 80% резерва, но лишь 20% бюджета

времени на празднование дней рождения, можете взять оставшиеся там часы и передать их каким-то более важным задачам.

**В** Можно ли организовать такую защищенную среду для выполнения программы, чтобы в ней не могли развиваться хитрые сообщения-черви, использующие человеческую психологию?

Похоже, что я без толку продолжаю повторять «обновите свои антивирусы» и «не запускайте произвольные вложения в письмах», потому что люди все равно *не обновляют* антивирусные программы и *запускают* случайные вложения.

Казалось бы, можно просто запретить выполнение двоичных файлов, но некоторые сначала сохраняют их, а потом запускают. Кроме того, если в письме говорится, что в нем содержится обновление системы защиты или игра, людям очень хочется их запустить.

Проблема в том, что невозможно полагаться на здравый смысл людей, если речь идет о том, какие файлы безопасно запускать на их машине.

Можно ли создать такую «клетку», чтобы черви были настолько изолированы (или явно помечены как вредоносные), чтобы значительное их размножение стало маловероятным?

*– Эрик Сеппанен*

**О** Это старая задача о создании «песочницы». Краткий ответ: скорее всего, нельзя.

Недавно я перешел на обслуживание своего сотового телефона в Sprint PCS, потому что их телефоны можно программировать на Java. У меня была масса идей по поводу приложений, которые можно написать:

1. Приложение для синхронизации справочника абонентов в моем телефоне со списком абонентов у меня в Outlook.
2. Приложение, позволяющее легко переключаться между «режимами» определенных настроек звонков и голосовой почты. Я мог бы, например, сказать телефону, что я в гимнастическом зале, и он стал бы сообщать в режиме голосовой почты «пожалуйста, оставьте голосовое сообщение, я получу его в течение часа», или сказать телефону, что я в подземке, и он стал бы сообщать «я в подземке и получу ваше сообщение через 20 минут».

Тогда я стал изучать J2ME и обнаружил, что 1) у меня нет доступа к телефонным номерам, хранящимся в аппарате, 2) у меня нет доступа к на-

стройкам телефона типа громкости звонка, 3) я не могу делать исходящие звонки, 4) я не могу пользоваться встроенным в телефон GPS для определения своего местонахождения, 5) у меня нет доступа к встроенной в телефон видеокамере, 6) и т. д. и т. д.

В принципе это отражает отношение Sun к защищенной среде исполнения, которое отдает предпочтение безопасности, даже если в результате ваша среда разработки оказывается совершенно бесполезной. С J2ME случится то же самое, что произошло с апплетами в броузере: единственное, что позволяет защищенная среда, это писать очень медленные игры (Ms. Pac-Man еще прилична, хотя и загружается 53 секунды), и эта технология практически бесполезна для каких-либо других целей.

Отношение Microsoft к защищенной среде совершенно иное: неизмеримо более опасное, но зато более интересное. На конференции PDC показали презентацию, на которой некто с помощью сотового телефона с .NET и примерно 10 строк кода мобильной .NET сделал фотографию, получил через GPS координаты и преобразовал их в городской адрес, записал звук и некоторые комментарии, а потом отправил все это в базу данных (пресловутое «приложение для страхового агента»). И все это нельзя сделать с помощью J2ME, несмотря на наличие соответствующих возможностей в аппарате.

Теперь о том, почему я сказал, что эта старая задача о создании «песочницы» известна со времен появления апплетов. Когда впервые появились апплеты, много было досужих рассуждений Джорджа Гильдера по поводу того, что новый текстовый процессор окажется просто большим апплетом Java. Беда была в том, что апплеты Java не могли читать или писать файлы на жесткий диск. Все происходило в защищенной среде. Для того чтобы обеспечить постоянное хранение каких-то данных, апплет должен был записывать файлы на жесткий диск, который находился неизвестно где. Кто-то решил, что можно разрешить хранить данные *внутри* песочницы, лишь бы только не было доступа к данным *вне* ее. Беда в том, что ваши данные как раз находятся *вне*. Но не обращайте на это внимания. Даже в такой замечательной конструкции, когда все ваши данные находятся *внутри* песочницы, хорошо ли вы защищены? Ведь все, что вас интересует, находится внутри песочницы. Следует признать: если вам нужна возможность выполнить произвольный код для редактирования своей графики, например, чтобы приставить голову Билла Клинтона к телу Марки Марка, то вам придется иметь возможность выполнить любой код, который редактирует ва-

шу графику, а если этот код решит попортить ваши картинки, то останется только посочувствовать вам.

Microsoft и Sun параллельно движутся в сторону «мелкозернистой» модели системы защиты, в которой можно более точно задавать для отдельных приложений их права. Ms. PacMan может читать и писать файлы, но это разрешение относится только к таблице лучших результатов, а обращаться к Сети программа, конечно, не имеет права. Политику безопасности можно усложнять бесконечно.

Как скажет вам любой, кто немного разбирается в защите данных, эта политика не будет эффективна. Чем сложнее система защиты, тем больше вероятность ее неверной настройки. Человек может справиться лишь с некоторым ограниченным уровнем сложности, и ни у кого нет времени на то, чтобы осуществлять тонкую настройку прав для всех своих приложений.

Поэтому в целом ситуация выглядит уныло. Скажем, мы никак не можем исключить, что бандит ударит пожилую женщину по голове бейсбольной битой и похитит у нее кошелек. Мы можем грозить бандитам наказанием и сажать их в тюрьму, но уверяю вас, что в настоящее время, если кто-то твердо решит ударить пожилую женщину по голове бейсбольной битой, ему это, наверное, удастся. Точно так же ничто не помешает вам съехать на своей машине в пропасть. Достаточно сесть в машину, заехать на скалу и съехать вниз – ничто вас не остановит.

Людям кажется, что в мире компьютеров жизнь должна быть намного безопаснее, чем в реальном мире, и они надеются, что там можно «защитить себя от самого себя», но этого никогда не будет, так же как производители автомобилей никогда не придумают машин, которые откажутся съезжать с горы в пропасть, а маленькие старушки не станут ходить по улицам в касках. И тем не менее жизнь продолжается, и лично я постараюсь найти что-то другое, из-за чего можно лишиться сна.

**В Что за шум вокруг списывания опционов на акции в расходы? Вроде бы, идея дать служащим возможность участвовать в прибылях компании неплоха. Почему списывание в расходы делает их менее привлекательными?**

*– Джейсон*

О Предположим, что у вас есть большая софтверная компания, цена акций которой каждый год вырастает на 100%. Кинув своим служащим несколько опционов на акции, вы можете обеспечить им эквивалент

тысяч и миллионов долларов в компенсацию жалованья, не потратив при этом никакой наличности. На самом деле можно даже одновременно снизить их основное жалованье, и они не станут возражать, потому что опционы на акции приносят им огромные деньги. И действительно, в конце 1980-х зарплаты в Microsoft были процентов на 30 или 40 ниже, чем у конкурентов, – за эти годы все, кто там тогда работал, стали миллионерами.

Откуда берутся все эти опционы, если они не стоят вам никаких наличных денег? Обычно выпускаются дополнительные акции. В результате во владении уже существующих акционеров оказывается меньшая доля капитала. Пусть, например, в обращении уже есть миллион акций, а вы выпускаете еще один миллион акций с целью дать своим служащим возможность выкупить эти акции по цене ниже рыночной (когда они воспользуются своими опционами), так что теперь в обращении оказывается два миллиона акций. Если стоимость компании не изменится, то каждая акция станет вдвое дешевле. Фактически вы берете деньги из карманов своих акционеров, чтобы расплатиться со своими служащими.

Почему вообще закон допускает такое? Он и не допускает, если только сами акционеры не согласятся на это, а они соглашаются, потому что с точки зрения акционера можно либо заплатить служащим, взяв деньги с банковского счета компании, что уменьшит стоимость каждой акции, либо заплатить служащим, увеличив количество акций, находящихся в обращении, что уменьшит стоимость каждой акции.

Уловили? Следите внимательно – я приведу еще один пример. Очень важно это понимать. Внимательно следите? Хорошо.

Я – большая софтверная компания, которая стоит два миллиона долларов. В обращении находится один миллион акций. Каждая акция поэтому стоит $2.

Теперь мне надо расплатиться со своими служащими. На зарплату нужен $1 000 000.

**Вариант 1:** я плачу наличными с банковского счета. Моя компания, стоившая два миллиона, стала стоить один миллион по определению, ведь мы только что выплатили 1 миллион наличными. Когда этот миллион ушел, стоимость компании должна уменьшиться на миллион, так? Поэтому цена каждой акции упала с 2 до 1 доллара.

**Вариант 2:** вместо выплат наличными я заручаюсь одобрением акционеров и выпускаю еще один миллион акций, раздав их служащим. Теперь в обращении находятся два миллиона акций вместо одного. Но компания

по-прежнему стоит два миллиона, потому что я не потратил никаких денег – за пределы компании ничего не ушло. Поэтому цена каждой акции снижается с 2 до 1 доллара просто потому, что их стало вдвое больше.

Обратили внимание на общее? В обоих случаях цена акций упала с 2 до 1 доллара.

Какое различие между вариантами 1 и 2?

- В обоих вариантах служащие получают 1 млн. долларов.
- В обоих вариантах акционеры видят, как цена их акций падает на 50 процентов – с 2 до 1 долл. за штуку.
- По варианту 2 компании, теоретически, должно быть лучше, потому что в ее распоряжении оказывается больше наличности.

А теперь решающее различие:

- По прежним правилам бухгалтерской отчетности, если не надо списывать в расход опционы, то по сценарию 2 в прибыли появляется лишний миллион по сравнению со сценарием 1.

До сих пор мы вообще не говорили о прибыли. Прибыль – это доходы минус расходы. Доходы у нас одинаковы в обоих сценариях. Единственная разница в том, что в одном случае миллион долларов ушел за пределы компании, а в другом нет.

Поскольку сценарии 1 и 2 практически одинаковы во всех отношениях, не совсем справедливо, что компании, следующие сценарию 2, могут показать в отчете лишний миллион долларов, которого нет у компании, выбравшей сценарий 1.

Иными словами, в обоих сценариях мы поступаем практически одинаково как в отношении служащих, так и в отношении акционеров. Так почему же во втором случае прибыль должна быть на миллион долларов больше? Нелепо, что вы вдруг становитесь «прибыльнее», потому что решили заплатить служащим за счет «разводнения» капитала, а не опустошения счета в банке. Для акционера результат одинаков.

Старая система бухгалтерской отчетности приводила к тому, что вычисленным Уолл-стритом показателям «доходности акций» нельзя было верить. Она была выгодна многим компаниям, которые выглядели очень прибыльными в смысле дохода на одну акцию.

Когда инвесторы сравнивают компании, они смотрят на такие показатели, как отношение цены к прибыли (цена одной акции, деленная на доход по одной акции). Если потребовать, чтобы компании записывали опционы акций в расход, то это будет означать, что их прибыль сократится на ту сумму, которой они разбавили имеющиеся акции, выпустив новые,

и тогда компании станут сообщать об одинаковых доходах на одну акцию независимо от того, первым или вторым сценарием они воспользовались, что честнее по отношению к акционерам и позволяет точнее сравнивать две компании друг с другом. Это означает, что рыночным аналитикам не придется самим вычислять, учитывая разводнение акций в результате опционов служащим на дополнительно выпущенные акции, как сравнить компанию #1 с компанией #2: они могут просто смотреть на доходность и знать, что опционы акций для служащих не дают одной компании кажущегося превосходства перед другой. В прежние времена единственный способ честно сравнить прибыль на одну акцию для двух компаний заключался в том, чтобы проработать их годовые отчеты, выяснить, что опционы привели к разводнению акций, и, исходя из этой информации, попытаться выполнить самостоятельные пересчеты «доходности», которые часто оказывались неполными или запоздалыми. Можно не сомневаться, что рыночные аналитики не особенно и утруждали себя этим. Разумеется, толпа с «Мотли Фул» вполне довольствовалась сравнением коэффициентов цена/прибыльность для разных компаний, не подозревая о том, что опционы акций для служащих означали, что показатели прибыльности искажались самыми непредсказуемыми способами.

Учет опционов на акции в статье расходов приводит к тому, что разводнение акций, вызванное расплатой акциями со служащими, учитывается должным образом и позволяет акционерам видеть более точную картину эффективности деятельности компании.

**В Прошу прощения, если этот вопрос уже задавали, но что вы думаете о том, как может повлиять на Microsoft подключение всех пользователей дома или на работе по 100-мегабитным широкополосным каналам?**

У большинства пользователей отпадет потребность в Windows или Office, поскольку при такой скорости можно выполнять любые веб-приложения. И в программном обеспечении Microsoft тоже не будет необходимости.

Вот пример: я уже пользуюсь Hotmail для электронной почты, а если бы у меня был достаточно быстрый канал и удаленное устройство памяти, то мог бы пользоваться и сетевым текстовым процессором или электронной таблицей. В результате мои данные стали бы доступны мне из любого места так же, как и моя электронная почта Hotmail.

*– Рой*

О Не так быстро! Не надо лихо экстраполировать.

Прежде всего, я весьма сомневаюсь, что вы захотите долго работать в текстовом процессоре, полученном в виде веб-страницы, особенно при нынешнем состоянии HTML. Пользовательские интерфейсы, разработанные для IE 6.x, грубы и медлительны. И это неслучайно – я не думаю, что Microsoft *хочет*, чтобы пользователю было приятно работать с формами HTML.

Во-вторых, у многих есть высокоскоростные соединения, но я пока не встречал никого, кто хотел бы заниматься обработкой текста или электронными таблицами в веб-броузере. Можете не сомневаться, бизнес Microsoft Office процветает.

Даже если бы возникла гипотетическая угроза со стороны офисных программ, реализуемых через веб-броузер, можете не сомневаться, что Microsoft нашла бы себе место и в этом бизнесе

В начале 1990-х всем казалось, что IBM должна исчезнуть, потому что она не поняла PC и не смогла преодолеть свою ментальность, связанную с большими машинами. О чем люди забывают, так это о том, что большая и успешная компания падает очень *долго*, и во время этого падения у нее есть *годы*, чтобы обрести для себя новую опору. Microsoft сейчас находится в таком же положении. С какой бы фантастической теорией относительно гибели Microsoft вы ни выступили (веб-приложения, скоростное соединение с Интернетом, Linux, антитрестовское законодательство или черные вертолеты), вы не сможете опровергнуть того, что у Microsoft достаточно средств, чтобы продолжать функционировать еще лет пять, даже если ее доходы внезапно обратятся в нуль, и при этом не уволить ни единого программиста. А ее доходы не могут обратиться в нуль завтра же. Даже в самых кошмарных сценариях доходы могут начать падать, скажем, на 20% в год, поэтому они вполне могут продержаться еще лет десять, прежде чем начать подумывать об увольнениях.

Поэтому недостаточно назвать изменения в технологии, которые угрожают Microsoft; необходимо еще предположить, что она не сможет ничего найти в ответ на протяжении многих лет.

# Алфавитный указатель

## Н



## О



## П