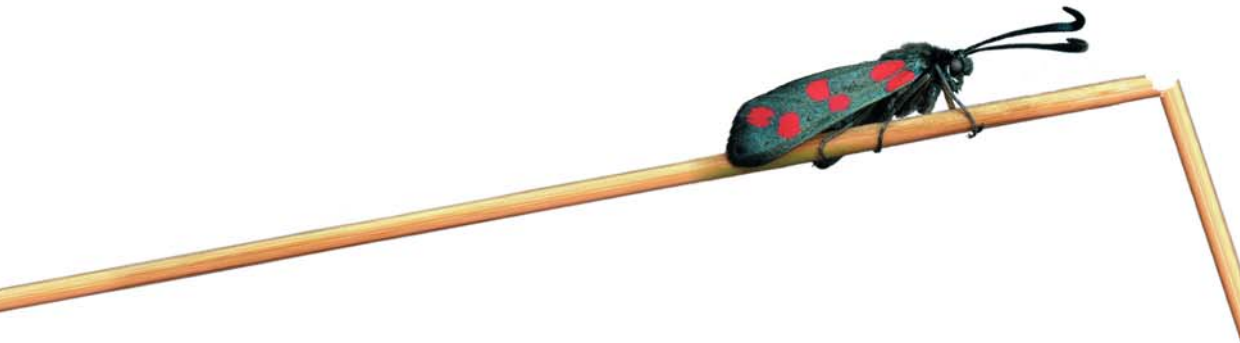
ПРОФЕССИОНАЛЬНО

Джоэл Спольски



ДЖОЭЛ И СНОВА О ПРОГРАММИРОВАНИИ

НОВЫЕ МЫСЛИ О 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®

MORE JOEL ON SOFTWARE

Further Thoughts 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: и снова о программировании

Новые мысли о 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
2009

Серия «Профессионально»

Джоэл Спольски

Джоэл: и снова о программировании

Новые мысли о 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*
Выпускающий редактор	*П. Щеголев*
Редактор	*Т. Темкина*
Корректор	*С. Минин*
Верстка	*Д. Орлова*

Спольски Дж.

Джоэл: и снова о программировании. Новые мысли о 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, 2009. – 320 с., ил.

ISBN 978-5-93286-144-8

Продолжение вышедшего в 2006 году бестселлера «Джоэл о программировании» представляет собой подборку самых популярных статей, опубликованных автором на его сайте http://www.joelonsoftware.com. Исключительный писательский талант, техническая эрудиция и язвительный ум Джоэла создали ему высочайшую профессиональную репутацию и принесли его сайту скандальную известность.

В книге затронуты разнообразные вопросы, касающиеся разработки и проектирования программного обеспечения, управления софтверным бизнесом, эффективного поиска и привлечения высококлассных сотрудников, организации рабочего места и общения с заказчиками. Автор предлагает практические советы как программистам, так и тем, кто руководит их работой.

ISBN 978-5-93286-144-8
ISBN 978-1-4302-0987-4 (англ)

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, www.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 25.12.2009. Формат 70x90$^{1}/_{16}$. Печать офсетная.
Объем 20 печ. л. Тираж 1500 экз. Заказ №
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Джареду

כי אהבת נפשו, אהבו

Оглавление

Об авторе

Джоэл Спольски – всемирно признанный эксперт по технологии разработки программного обеспечения. Его веб-сайт *Joel on Software* (*www.joelonsoftware.com*) популярен среди разработчиков всего света и переведен более чем на 30 языков. Основатель нью-йоркской компании Fog Creek Software, он создал FogBugz – популярную среди программистов систему управления проектами. Ранее Джоэл работал в Microsoft, где как член команды Excel разрабатывал VBA, а в компании Juno Online Services он занимался разработкой интернет-клиента Juno, применяемого миллионами пользователей. Он автор книг «User Interface Design for Programmers» (Проектирование пользовательского интерфейса для программистов), Apress, 2001, «Joel on Software»[1], Apress, 2004 и «Smart and Gets Things Done» (Руководство Джоэла Спольски по подбору программистов и управлению ими), Apress, 2007, а также составитель сборника «The Best Software Writing I»[2], Apress, 2005. Джоэл – выпускник Йельского университета с дипломом по вычислительной науке. Служил в воздушно-десантных войсках Израиля и был одним из основателей кибуца Ханатон.

---

[1] Джоэл Спольски «Джоэл о программировании». – Пер. с англ. – СПб.: Символ-Плюс, 2006.

[2] Джоэл Спольски «Лучшие примеры разработки ПО». – Пер. с англ. – СПб.: Питер, 2006.

Джоэл, APRESS, блоги и блуки

«Давным-давно, в далекой-далекой галактике...» На самом деле, это случилось в конце 2000 года – первого полного календарного года работы Apress. Наше крошечное, еще не ставшее известным издательство компьютерной литературы в тот год собиралось выпустить лишь горстку книг – примерно столько, сколько сейчас у Apress выходит в месяц.

Я тогда только осваивал нелегкое издательское дело и, пожалуй, больше, чем следовало, бродил по веб-сайтам и программировал. Однажды я попал на сайт «Joel on Software», автор которого отличался резкими суждениями, необычным, умным стилем письма и явной склонностью бросать вызов общепринятым мнениям. Например, он писал о том, как плох обычно бывает пользовательский интерфейс – главным образом из-за того, что программисты, в общем-то, не знают о потребностях пользователей ничего – «bupkis», как сказали бы мы с Джоэлом на знакомом нам обоим нью-йоркском идише. И я, как многие, подсел на эти и другие статьи Джоэла.

Вдруг меня озарило: ведь я же издатель, и раз мне нравится то, что он пишет, почему бы не сделать из этого книгу? Я познакомился с Джоэлом, и хотя поначалу он был настроен скептически, убедил его, что если сделать из статей по интерфейсам книгу, народ набросится на нее, и мы с ним заработаем кучу денег. (Разумеется, это было задолго до успеха FogBugz и того времени, когда он стал неплохо получать за свои выступления, – но тогда мы оба были моложе и, что скрывать, гораздо беднее.)

Джоэл немного расширил материал, чтобы книга стала привлекательнее и (моя мысль) лучше продавалась. Поскольку у Apress не было опыта полноцветной печати, пришлось разбираться и с этим. 21 июня 2001 года – официальная дата выхода книги «User Interface Design for Programmers» (Советы программистам по проектированию пользовательского интерфейса), ныне признанной первым в мире «блуком»[1]. Совершенно неожиданно для меня и других издателей компьютерной литературы она стала бестселлером, по меркам того времени. Кстати, она до сих пор допечатывается, хорошо продается, и ее стоит прочесть. (Да, уже не как друг, а как издатель: Джоэл, как насчет новой редакции?)

Сегодня некоторые утверждают, что книга «User Interface Design for Programmers» – не настоящий блук, поскольку включает «слишком много» нового материала, которого не было на веб-сайте Джоэла, представляя собой некий гибрид, – но, думаю, не стоит забывать о том, что это была первая подобная книга.

Спустя несколько лет «Joel on Software» стал самым популярным в мире блогом для программистов, потому что Джоэл продолжал писать удивительно интересные статьи; пожалуй, можно назвать классической самую знаменитую из них – «How Microsoft Lost the API War» (Как Microsoft проиграла войну API), которая буквально вызвала коренной переворот в некоторых разработках Microsoft.

И тут меня вновь озарило: надо собрать лучшие из этих статей и издать как есть, добавив лишь комментарии Джоэла там, где он сочтет это уместным. Несмотря на то что практически весь материал новой книги, названной «Joel on Software», уже был опубликован в Сети, и никто не понимал, зачем Apress издавать ее в конце 2004 года, эта книга переиздавалась десять раз и по сей день остается бестселлером.

Мне кажется, секрет здесь в том, что смаковать деликатес вроде статьи Джоэла гораздо приятнее в печатном виде, чем в окне броузера.

А Джоэл продолжал размышлять о том, как писать хорошие программы или нанимать хороших программистов, не переставая бросать вызов общепринятым мнениям. И я убедил его, что пора издать сиквел, собрав в нем лучшее из опубликованного после выхода того сборника 2004 года.

И вот перед вами второе собрание откровений, мыслей по поводу и, да, пламенных тирад Джоэла – все это в характерном для него блестящем сти-

---

[1] Блук (англ. blook, от blog – блог, сетевой журнал, или дневник, и book – книга) – книга на основе материалов блога. – *Прим. перев.*

ле. Текст в последнем выпуске «лучшего у Джоэла» не претерпел изменений, кроме мелкого редактирования, тем не менее его контрастность гораздо выше, чем на экране компьютера или даже устройства Kindle – того, что теперь называется блуком. (И мы с Джоэлом надеемся, что это собрание понравится вам не меньше первого.)

У этой книги, как и у первого сборника, довольно необычные обложка[1] и подзаголовок. Дело в том, что мы с Джоэлом библиофилы (вернее, Джоэл – библиофил, а я – библиоман), и нам очень нравится, как книгоиздатели XVII и XVIII веков оформляли свои книги и их названия. Обложка первой книги «Joel on Software» – дань уважения «Анатомии меланхолии» Бертона, а здесь мы отдали должное «Левиафану» Гоббса, на знаменитом фронтисписе которого – гигант, составленный из множества человеческих фигурок, что показалось нам с Джоэлом неплохой метафорой программирования: все вместе создают нечто огромное, но ключевую роль играет индивидуум.

И под конец, личное замечание. Несмотря на всю свою нынешнюю известность, Джоэл – все тот же человек практического склада, или, на нашем с ним диалекте, настоящий «mensch», близкой дружбой с которым я горжусь.

Гэри Корнелл,
соучредитель Apress

[1] Имеется в виду обложка оригинального английского издания – *Прим. перев.*

ЧАСТЬ ПЕРВАЯ

Как управлять людьми

ГЛАВА ПЕРВАЯ

Мое первое «рецензирование БиллГ»

16 июня 2006 года, пятница

В допотопные времена в Excel был очень неудобный язык программирования, которому не дали названия. Мы называли его «макросы Excel». Он был весьма ограничен – без переменных (значения приходилось хранить в ячейках таблицы), локальных переменных и вызовов подпрограмм – короче, сопровождать такой язык было практически невозможно. В нем имелись некоторые более сложные функции вроде `Goto`, но метки были скрыты от глаз.

Единственное, что его оправдывало, это явное превосходство над макросами Lotus, представлявшими собой лишь длинную строку последовательности нажатий клавиш, вводимую в ячейку таблицы.

17 июня 1991 года я приступил к работе в Microsoft в составе команды Excel. Моя должность называлась Program Manager. Предполагалось, что я как-то решу проблемы языка. Подразумевалось, что это решение будет связано с языком программирования Basic.

Basic? Фу!

Некоторое время я вел переговоры с разными группами разработчиков. Только что появился Visual Basic 1.0, выглядевший довольно круто. Также имелся неудачный проект под кодовым названием MacroMan и еще один проект – создание объектно-ориентированного языка Basic – под кодовым названием Silver. Команде Silver было сказано, что первый клиент для их продукта уже есть – это Excel. Менеджер по маркетингу проекта Silver Боб Уаймен (Bob Wyman) – да, тот самый – мог продать свою технологию только мне.

Проект MacroMan, как я сказал, был неудачным, и с некоторым трудом его все же удалось закрыть. Команда Excel убедила команду Basic в том, что ей нужен особый вариант Visual Basic для Excel. Моими стараниями в Basic включили четыре вещи, которые мне очень нравились. Был добавлен структурный тип данных Variant, позволяющий хранить данные любого типа, потому что иначе сохранить в переменной содержимое ячейки рабочего листа можно было только с помощью оператора переключения. Также было добавлено позднее связывание, ныне известное как IDispatch, или COM-автоматизация, потому что исходный проект Silver требовал глубокого понимания системы типов, чего трудно было ожидать от тех, кто станет писать макросы. И еще я получил две свои любимые синтаксические конструкции – For Each, украденную из csh, и With, украденную из Pascal.

Потом я засел за написание спецификации Excel Basic – огромного документа, разросшегося до сотен страниц, – думаю, в конце их стало не меньше пятисот. («Каскадное проектирование» – усмехнетесь вы. Да-да, приятель, оно самое.)

В те времена существовало понятие «рецензирования БиллГ». Практически любой крупной и важной функции требовалась оценка Билла Гейтса. Мне сказали послать в его офис на рецензию экземпляр спецификации (фактически, это целый том лазерных распечаток).

Я тут же напечатал и отослал ее.

В тот день у меня оставалось немного свободного времени, и я стал размышлять над тем, достаточно ли в Basic функций для работы с датами и временем, чтобы делать с ними все то, что позволяет Excel.

В большинстве программных сред даты хранятся в виде действительных чисел. При этом целая часть – это количество дней, истекших с некой оговоренной даты в прошлом, называемой началом эпохи (epoch). В Excel сегодняшняя дата 16 июня 2006 года хранится как число 38884, за точку отсчета принято 1 января 1900 года.

Я стал экспериментировать с различными функциями даты и времени Basic и Excel и обнаружил в документации Visual Basic нечто странное: в Basic началом эпохи считалось 31 декабря 1899 года, а не 1 января 1900 года, но по каким-то причинам сегодняшняя дата и в Excel, и в Basic представлялась одинаково.

Что?!

Я стал искать разработчика Excel, достаточно старого, чтобы помнить такие вещи. Похоже, ответ знал Эд Фрайз (Ed Fries).

– Ха, – сказал он мне, – проверь 28 февраля 1900 года.

– 59, – отвечал я.

– Теперь 1 марта 1900 года.

– 61!

– А где 60? – спросил Эд.

– 29 февраля 1900 года, год был високосным! Он делится на 4!

– Мысль интересная, но неверная, – сказал Эд, оставив меня в недоумении.

Пришлось провести некоторые исследования. Оказалось, что годы, которые делятся на 100, *бывают* високосными, только если при этом еще делятся на 400.

1900-й год не был високосным.

– В Excel ошибка! – воскликнул я.

– *Не совсем* так, – возразил Эд. – Нам пришлось пойти на это, чтобы импортировать таблицы Lotus 1-2-3.

– Значит, ошибка в Lotus 1-2-3?

– Да, но, скорее всего, умышленная. Требовалось уместить Lotus в 640 Кбайт – не так много памяти. Если не обращать внимания на 1900 год, то можно проверить год на високосность по двум правым разрядам числа – они должны быть нулевыми. Быстро и легко. Наверное, ребята из Lotus решили, что ничего не случится, если какие-то два месяца в далеком прошлом будут считаться неправильно. Похоже, разработчики Basic дотошно учли эти два месяца, сдвинув эпоху на день назад.

– Вон оно что, – сказал я и стал выяснять, что означает флажок `1904 Date System` (Система дат 1904) в окне параметров.

На следующий день должно было состояться рецензирование БиллГ.

30 июня 1992 года.

В те дни в Microsoft было гораздо меньше бюрократии. Это сейчас там одиннадцать или двенадцать уровней управления, а тогда я подчинялся Майку Конте (Mike Conte), который подчинялся Крису Грэму (Chris Graham), который подчинялся Питу Хиггинсу (Pete Higgins), который подчинялся Майку Мэйплзу (Mike Maples), который подчинялся Биллу. Всего примерно шесть уровней иерархии. Мы еще посмеивались над компаниями вроде General Motors, где этих уровней было восемь или около того.

На моем рецензировании БиллГ присутствовала вся иерархическая цепочка вместе с родными и двоюродными сестрами и тетками, а также

человек из моей команды, который должен был точно подсчитать, сколько раз Билл скажет f... (чем меньше, тем лучше).

Вошел Билл.

Я подумал: как странно, что у него тоже две ноги, две руки, одна голова и так далее – совсем как у обычных людей.

В руках у него была моя спецификация.

В руках у него была моя спецификация!

Он сел, обменявшись с кем-то из незнакомых мне управленцев шуткой, смысл которой до меня не дошел. Несколько человек рассмеялись.

Билл повернулся ко мне.

Я заметил комментарии на полях моей спецификации. Он прочел первую страницу!

Он прочел первую страницу моей спецификации и что-то написал на ее полях!

Если учесть, что мы послали ему спецификацию всего 24 часа назад, он должен был читать ее накануне вечером.

Он стал задавать вопросы. Я отвечал. Вопросы были достаточно простыми, но я не в силах их вспомнить, потому что неотрывно следил за тем, как он листает спецификацию...

Он листал спецификацию! [Успокойся, ты же не маленькая девочка!]

И НА ВСЕХ ПОЛЯХ БЫЛИ ПОМЕТКИ. НА КАЖДОЙ СТРАНИЦЕ. ОН ПРОЧЕЛ ВСЮ ЭТУ ЧЕРТОВУ СПЕЦИФИКАЦИЮ И НАПИСАЛ ЗАМЕЧАНИЯ НА ПОЛЯХ.

Он Прочел Все! [БОЖЕ МИЛОСТИВЫЙ!]

Вопросы становились более сложными и детальными.

Они казались несколько случайными. Я уже был готов считать Билла своим приятелем. Какой славный малый! Он прочел мою спецификацию! Он, наверное, хочет задать мне несколько вопросов, связанных с заметками на полях! Я занесу все его замечания в систему контроля ошибок и прослежу, чтобы все они были учтены, и как можно скорее!

И вот решающий вопрос.

– Не знаю, ребята, – сказал Билл, – есть здесь кто-нибудь, кто *действительно* в деталях разбирается, как это сделать? Например, все эти функции даты и времени. В Excel их очень много. Будут ли такие же функции в Basic? И будут ли они работать точно так же?

– Да, – ответил я, – кроме января и февраля 1900 года.

Тишина.

«Счетчик f» и мой босс изумленно переглянулись. *Откуда он это взял? Января и февраля КАКОГО ГОДА?*

– ОК. Хорошая работа, – сказал Билл. Он взял свой размеченный экземпляр спецификации.

...Погодите! Я же хотел...

И вышел.

– Четыре, – объявил «счетчик f», и все заговорили, что такого низкого счета на их памяти не было и что Билл с годами становится мягче. Ему тогда было 36.

Позже мне все объяснили. «На самом деле, Билл не собирается обсуждать твою спецификацию, ему просто нужно убедиться, что ты владеешь темой. Его обычный стиль – задавать вопросы все сложнее и сложнее, пока он не уличит тебя в каком-нибудь незнании, и тогда можно сделать тебе выволочку за неподготовленность. Никто не знал, что произойдет, если кто-то ответит на самый трудный его вопрос, потому что такого раньше не случалось».

– Если бы на этом совещании был Джим Манци, – сказал кто-то, – он спросил бы, что такое функция даты.

Джим Манци (Jim Manzi) – это тот менеджер с MBA, задачей которого было загнать в гроб Lotus.

Замечание было верным. Билл Гейтс поразительно разбирался в технических деталях. Он понимал, что такое Variant, COM-объекты, IDispatch, и чем автоматизация отличается от vtables, и как возникают двойственные интерфейсы. Его интересовали функции даты. Он не влезал в программу, если доверял тем, кто над ней работает, но его нельзя было провести на мякине, потому что он сам был программистом. Настоящим, действующим программистом.

Смотреть, как непрограммист пытается управлять софтверной компанией – все равно что наблюдать за новичком-серфингистом.

«Все отлично! У меня на берегу отличные помощники, которые подсказывают мне, что нужно делать!» – говорит он и снова падает в воду. Типичное поведение администратора, считающего управление универсальной функцией. Не повторит ли Стив Балмер подвиг Джона Скалли, который едва не довел Apple до исчезновениия, хотя совет директоров посчитал, что торговля пепси-колой – подходящая подготовка для руководителя компьютерной фирмы? Поклонники MBA склонны полагать, что справятся с управлением организацией, в работе которой они не разбираются.

С годами Microsoft разрослась, пребывание Билла у руля чересчур затянулось, а некоторые сомнительные с этической точки зрения решения вынудили руководство уделить слишком много внимания противостоянию с правительством США. Стив взял на себя роль CEO, теоретически для того, чтобы Билл смог больше времени посвятить тому, в чем он всего сильнее, – управлению организацией, разрабатывающей программное обеспечение, – но незаметно, чтобы в результате разрешились проблемы компании, обусловленные наличием тех самых одиннадцати уровней управления, вечными непрерывными совещаниями, упрямым стремлением создавать все мыслимые продукты (сколько миллиардов долларов Microsoft ушли на исследования и разработки, юридические услуги, поддержку репутации – только потому, что было решено не просто сделать веб-броузер, но и распространять его бесплатно?) и двумя десятилетиями безответственного и поспешного найма новых работников, в результате чего резко упал средний интеллектуальный уровень сотрудников Microsoft (Дуглас Купленд (Douglas Coupland) в «Microserfs»: «Только в 1992 году они приняли на работу 3100 человек, и можете не сомневаться, что не все из них оказались самородками»).

Ну, ладно. Все идет своим путем. Excel Basic превратился в Microsoft Visual Basic for Applications for Microsoft Excel с таким обилием ™ и ®, что я не знаю, как их все тут разместить. Я покинул эту компанию в 1994 году, полагая, что Билл совершенно забыл про меня, пока мне не попалось короткое интервью с ним в «Wall Street Journal», в котором он отметил, почти мимоходом, что очень трудно было найти, скажем, хорошего руководителя проекта для Excel. Дескать, такие люди на дороге не валяются, или что-то в этом роде.

Может, он имел в виду меня? Да нет, скорее, кого-то другого.

А впрочем, кто его знает.

ГЛАВА ВТОРАЯ

Как искать выдающихся разработчиков

6 сентября 2006 года, среда

Куда подевались выдающиеся разработчики?

У вас впервые открылась вакансия, и вы пошли обычным путем, размещая повсюду объявления, изучая крупные сетевые сайты и получая тысячи резюме.

Просматривая их, вы замечаете: «Хе, этот может подойти», или: «Ни в коем случае!», или: «А смогу ли я убедить его переехать в Буффало?» Но вот чего *никогда* не случится – могу дать голову на отсечение, – так это что вы воскликнете: «Вот это специалист, просто золото! Мы должны получить его любой ценой!» На самом деле, вы можете прочесть тысячи резюме – если допустить, что вы знаете, как их читать (а это непросто), – и тысячи объявлений о поиске работы, но так никогда и не найти выдающегося программиста. Ни одного.

И вот почему.

Выдающиеся программисты – да и лучшие специалисты в любой отрасли – просто *никогда не появляются на рынке*.

В среднем, выдающийся программист *может* сменить за свою карьеру примерно четыре места работы.

Тот, что блестяще окончил колледж, попадает на стажировку благодаря своему профессору, имеющему связи с производством, а затем получает предложение от этой компании, даже еще не начав искать работу. Если он и уйдет из компании, то, скорее всего, только затем, чтобы открыть собст-

венное дело на пару с приятелем, или перейдет в другую компанию вместе с боссом, или ему очень захочется поработать, скажем, с Eclipse, потому что Eclipse – это круто, и он ищет работу с Eclipse в BEA или IBM, и, конечно, получит ее, потому что это действительно блестящий специалист.

Если вам *повезет* – по-настоящему *повезет*, то нужный вам человек может мелькнуть на свободном рынке – например, если его жена решит пройти интернатуру в Анкоридже, и он пошлет свое резюме в те немногие места в Анкоридже, которые сочтет достойными своего внимания.

Но большинство видных разработчиков (и это почти тавтология)... заметны среди других (ОК, пусть будет тавтология), и будущие наниматели быстро замечают их выдающиеся качества, а это значит, что они, в сущности, могут работать там, где захотят, поэтому им нет нужды рассылать массу резюме или предлагать свои услуги во многих местах.

Хотели бы вы принять на работу такого человека? Наверняка.

Следствием закона, по которому выдающихся специалистов нет на рынке, является то, что рынок *переполнен* плохими – совершенно неквалифицированными – работниками. Их постоянно увольняют, потому что они не справляются со своими обязанностями. Их компании терпят неудачи – обычно из-за того, что кроме них в такой компании еще куча горе-программистов, – но иногда они оказываются *настолько неквалифицированными, что способны довести компанию до краха и в одиночку*. Да, бывает и так.

К счастью, таких на редкость неподготовленных людей редко берут на работу, но, продолжая искать ее, они идут на *Monster.com* и регистрируются на 300 или 1000 вакансий сразу в надежде выиграть в этой лотерее.

Выдающихся работников очень мало, и они никогда не появляются на рынке рабочей силы, в то время как некомпетентные работники, хотя и встречаются почти *столь же редко*, в течение своей трудовой карьеры пытаются устроиться в тысячи разных мест. Так что – возвращаясь к той кипе резюме, которые вы набрали на Craigslist, – не стоит удивляться, что большинство из них не от тех, кого вы хотели бы принять на работу.

Внимательный читатель, скорее всего, заметит, что я еще ничего не сказал о самой большой группе – о надежных и компетентных работниках. Их на рынке больше, чем выдающихся, но меньше, чем некомпетентных, и в целом *несколько* их резюме найдется в вашей пачке из 1000 штук, но обычно такая пачка, которая есть у каждого менеджера по подбору персонала в Пало-Альто, окажется все тем же набором из 970 резюме от тех же

970 некомпетентных, кто подает заявление на все вакансии, какие только есть в Пало-Альто, и может делать это всю жизнь, и только 30 резюме, которые вообще стоит рассматривать, и лишь изредка одно-два из них принадлежат выдающимся программистам. Найти эту иголку в стоге сена можно, но, как мы увидим, нелегко.

Можно их как-то найти?

Да!

Пожалуй, Можно!

Скажем так, Если Повезет.

Вместо процедуры «собрать резюме/отсеять резюме» следуйте процедуре «выследить лидеров и найти способ поговорить с ними».

У меня для этого есть три главных метода:

1. Поход в горы.
2. Производственная практика.
3. Создание собственного сообщества.*

(«Создание собственного сообщества» помечено у меня звездочкой как задача повышенной трудности – вроде той знаменитой математической проблемы, которую Джордж Данциг решил, потому что опоздал на занятие и не знал, что она считается неразрешимой.)

Конечно, у вас могут возникнуть собственные идеи. Я расскажу о том, что приносит успех мне.

На гору, Дживз!

Подумайте, где можно встретить людей, которых вы хотите принять на работу. В каких конференциях они участвуют? Где они живут? В какие организации входят? На каких веб-сайтах бывают? Не закидывайте большую сеть на *Monster.com*, лучше на сайте «Joel on Software» зайдите на форум, посвященный работе, ограничив поиски теми умными людьми, которые читают мой сайт. Посетите действительно интересные технические конференции. Выдающихся разработчиков для Маков вы найдете у Apple на WWDC. Выдающиеся программисты для Windows будут у Microsoft на PDC. Есть несколько хороших конференций open source.

Посмотрите, какие новые технологии сегодня в моде. В прошлом году это был Python, в этом – Ruby. Побывайте на соответствующих конфе-

ренциях, и вы найдете там людей, которые интересуются новинками, любят быть первопроходцами и стремятся все усовершенствовать.

Покрутитесь в коридорах, заговаривая со всеми, ходите на технические совещания и после приглашайте выступавших на кружку пива, и если обнаружите кого-то толкового – АГА! – тут уж дайте волю флирту и безудержной лести. «Оо-о, как это *интересно*! – говорите вы.– Невероятно, до чего ты *умен*. И красив, к тому же. Я забыл, где ты работаешь? *Где*? Хммм... А ты не думал, что для тебя нашлось бы место получше? По-моему, у меня в компании сейчас ищут специалиста...»

Следствие из этого правила: *не размещайте* свои объявления на крупных широкопрофильных сайтах поиска работы. Как-то летом я случайно поместил на MonsterTRAK наше предложение летней практики, заплатив немного больше обычного за то, чтобы наше предложение было видно в каждой школе США. В итоге мы получили буквально сотни резюме, ни одно из которых не прошло во второй круг. Мы потратили кучу денег, получив кучу резюме практически без шансов найти тех людей, которые были нам нужны. После нескольких дней такой работы мне пришло в голову, что MonsterTRAK – не то место, откуда мы могли бы получить резюме нужного нам кандидата. Аналогично, когда Craigslist только появился, и его посетителями были пионеры интернет-индустрии, мы нашли замечательных людей, публикуя рекламу в Craigslist, но сегодня этим сайтом пользуются все, кто мало-мальски умеет обращаться с компьютером, поэтому оттуда поступает слишком много резюме, а соотношение иголка/сено слишком мало.

Производственная практика

Хороший способ перехватить выдающегося работника, которого нельзя найти на рынке труда, это найти его тогда, когда он еще и не подозревает о существовании такого рынка, – пока он еще учится.

Некоторые менеджеры против того, чтобы брать практикантов. Они считают, что те еще не сформировались и недостаточно квалифицированны. В известном смысле, это правда. У практикантов нет опыта, как у штатных работников (откуда ж ему у них взяться!). Они требуют больше внимания и только некоторое время спустя достигнут требуемого уровня. Правда, приятная особенность нашей профессии состоит в том, что многие выдающиеся программисты начинают писать программы в 10-летнем возрасте. И пока все их сверстники играют в футбол (это такая игра для тех,

кто не умеет программировать; в ней нужно ударять ногой по круглому объекту под названием «мяч» – довольно странное занятие), они, забравшись в комнату отца, пытаются скомпилировать ядро Linux. Вместо того чтобы бегать за девчонками, они яростно доказывают в телеконференциях полную ущербность тех языков программирования, в которых отсутствует механизм вывода типов наподобие имеющегося в Haskell. Они не организовывают рок-группу в своем гараже – в это время они мастерят крутой хак, который перевернет вверх тормашками все изображения на вебстраницах у соседа, коварно подключавшегося к их беспроводной точке доступа Wi-Fi. БУ-ГА-ГА!

Так что, в отличие от таких областей, как, скажем, право или медицина, на втором или третьем году обучения в колледже эти ребята уже могут быть очень неплохими программистами.

Наверное, каждому приходится подавать заявление о приеме на работу – хотя бы *одно*, первое, и большинство молодых людей задумывается об этом только на последнем году обучения. При этом они, как правило, не особо мудрствуя, пишут заявление только тогда, когда в учебном городке случается какая-нибудь кампания по найму. В хороших колледжах студенты обычно получают достаточно хороших предложений работы от рекрутеров, охотящихся в студенческих городках, чтобы искать других работодателей, не затрудняющих себя присутствием в кампусах.

Вы можете принять участие в этом безумии, найме на работу в кампусах, – в чем, поймите меня правильно, нет ничего дурного, – либо пойти против этой системы, попытавшись заполучить толковых ребят за год или два *до* окончания ими учебы.

Подобный метод помог мне достичь больших успехов в Fog Creek. Каждый раз процедура начинается в сентябре, когда с помощью всех доступных мне ресурсов я начинаю охотиться за лучшими студентами факультетов вычислительной науки, имеющихся в стране. Я рассылаю письма по добрым двум сотням факультетов Computer Science. Я просматриваю все списки студентов по специальности CS, которым на тот момент предстоит еще два года учиться (скорее всего, чтобы получить эти списки, вам понадобится свой человек на факультете – преподаватель или студент). Затем я пишу личное письмо каждому обнаруженному мной студенту по специальности CS. Не электронное, а настоящее письмо на бланке Fog Creek с моей подписью. Очевидно, что им нечасто приходят такие письма, поэтому они не оставляют их без внимания. Я сообщаю, что у нас есть места для практикатов, и предлагаю подать заявление. Я посылаю электронные

письма преподавателям CS и бывшим сокурсникам, которые обычно участвуют в каких-нибудь почтовых списках рассылки для обучающихся по специальности CS, куда они и пересылают мои письма.

В итоге мы получаем массу заявок на эту летнюю практику и можем выбирать. В последние годы я получал по 200 заявлений на каждое место практиканта. Обычно мы просеиваем эту кучу заявлений примерно до 10 на место и приглашаем оставшихся на телефонное интервью. Из тех, кто прошел телефонное собеседование, двоих-троих мы обычно приглашаем в Нью-Йорк для личной беседы, оплачивая расходы.

Ко времени личной беседы вероятность того, что тот или иной человек нам нужен, уже весьма высока, и тут начинается полноценная *вербовка*. В аэропорту их ждет лимузин с водителем в униформе, который, получив багаж, отвозит их в центр модного района, где по улицам в любое время суток разгуливают модели, в довольно крутой отель, которого они, возможно, в жизни не видели, а в ванной комнате там приспособления, как из Музея современного искусства (дай им бог справиться с зубной щеткой). В номере отеля кандидата ждет подарочный набор: фирменная футболка, маршрут пешеходной экскурсии по Нью-Йорку, составленный сотрудниками Fog Creek, и фильм о прошлогодней летней практике, уже заряженный в DVD-плеер, – и они тут же могут узнать, как приятно провели время их предшественники.

После собеседований, проводящихся в течение дня, мы предлагаем им остаться в Нью-Йорке на пару дней за наш счет, чтобы осмотреть город, а затем лимузин отвозит их обратно в отель, а после в аэропорт, откуда они возвращаются домой.

Даже если все проверки прошел только один из трех претендентов, добравшихся до личного собеседования, очень важно, чтобы у них остались положительные впечатления. Те, кто не прошел собеседование, вернутся в кампус с тем впечатлением, что мы – классный работодатель, и расскажут друзьям о шикарном отеле в Большом яблоке, и друзья подадут заявления на следующую летнюю практику – хотя бы ради приятного путешествия.

Собственно, во время этой летней практики студенты обычно задумываются: «OK, прекрасная работа на лето, полезный опыт и, *может быть*, чем черт не шутит, это закончится зачислением в штат». Мы смотрим немного дальше. Летом мы решаем, нужны ли они нам как постоянные работники, а они решают, хочется ли им работать в нашей фирме.

Мы даем им реальные задания. Трудные. Наши практиканты работают над реальным промышленным кодом. Иногда им достаются самые интересные новые задачи, чему могут завидовать некоторые постоянные сотрудники, но это жизнь. Однажды летом у нас было четыре практиканта, которые сделали совершенно новый продукт с нуля. Эта практика окупила себя за считанные месяцы. Даже если продукт, над которым они работают, не совсем новый, они пишут реальный код, поставляемый клиенту и обеспечивающий какую-нибудь важную функцию, за которую они полностью отвечают лично (разумеется, не без помощи опытных наставников).

Кроме того, мы стараемся, чтобы они хорошо провели время. Мы устраиваем вечеринки и дни открытых дверей. Мы бесплатно поселяем их в довольно милом местечке неподалеку, где они могут познакомиться со сверстниками из других компаний или учебных заведений. Каждую неделю у нас проводится какой-нибудь общественное мероприятие или культпоход: Бродвейский мюзикл (в этом году они были в восторге от «Avenue Q»), кинопремьера, музейная экскурсия, водная прогулка вокруг Манхэттена, посещение матча Yankees. Не поверите, но в этом году огромным успехом пользовалось «восхождение на скалу» (то есть на крышу какого-нибудь высотного здания на Манхэттене). Никто представить не мог, что это восхождение способно вызвать у них такой душевный подъем. В каждом таком мероприятии участвуют несколько сотрудников Fog Creek.

В конце лета всегда найдется несколько практикантов, сумевших убедить нас в том, что они из тех замечательных программистов, которых мы просто обязаны взять. Подчеркиваю: не всех. Кто-то из них – отличный программист, таких мы готовы принять, а прочие могут прекрасно проявить себя где-то еще, помимо Fog Creek. К примеру, в нашей компании сильно развито самоуправление и у нас мало менеджеров среднего звена, поэтому мы предполагаем, что наши сотрудники работают самостоятельно. За всю историю летней практики нам пару раз встречались практиканты, которые прекрасно работали под чьим-то руководством, но в Fog Creek, без достаточного контроля, они бы не справились.

Если мы действительно хотим кого-то принять на работу, то нет смысла тянуть с этим. Мы делаем предварительное предложение о приеме на полный рабочий день при условии, что сначала они закончат учебу. Это щедрое предложение. Мы хотим, чтобы они вернулись к себе в колледж, поговорили со своими друзьями и поняли, что предлагаемое им начальное денежное вознаграждение выше, чем у кого бы то ни было.

Не переплачиваем ли мы им? Нисколько. Дело в том, что при назначении жалованья принимаемому работнику обычно учитывают риск того, что он окажется бесполезен. Но мы уже проверили этих ребят и знаем, что они могут работать, так что риска нет. Мы знаем, на что они способны. Поэтому, принимая их на работу, мы знаем их лучше, чем любой другой предприниматель, у которого есть одни только данные собеседования. Следовательно, мы можем платить им больше. Мы лучше информированы, поэтому и готовы платить больше, чем работодатели, не владеющие такой информацией.

Если мы хорошо постарались, как обычно и бывает, то к этому моменту наш практикант легко принимает предложение. Иногда требуются дополнительные меры убеждения. Порой им трудно отказаться от других возможностей, но благодаря тому, что предложение Fog Creek остается в силе, когда им потребуется встать в 8 утра и надеть костюм для собеседования в Oracle, весьма вероятно, что, проснувшись по звонку будильника, они скажут себе: «А какого черта мне вставать в 8 утра, надевать костюм и идти в Oracle, если меня и так ждет отличная работа в Fog Creek?!» И, надеюсь, они так и не доберутся до этого собеседования.

Прежде чем продолжить, я должен дать некоторые разъяснения относительно практики в области вычислительной науки и разработки программного обеспечения. Сегодня в нашей стране работа практикантов повсеместно *оплачивается*, и выплачиваемое жалование обычно достаточно привлекательно. В различных других областях – от издательской деятельности до музыки – практика чаще бесплатная, мы же платим 750 долларов в неделю плюс оплачиваем проживание, обед и проезд на метро, не говоря об оплате переезда и прочих выгодах. Сумма в долларах несколько ниже средней, но зато бесплатно предоставляется жилье, поэтому в итоге условия несколько выше средних. Я решил сказать об этом, потому что всякий раз, когда я рассказываю о практике на своем сайте, кто-нибудь обязательно неправильно понимает меня и заключает, что я получаю прибыль за счет рабского труда или что-то в этом роде. Эй ты, салага! Ну-ка отожми мне холодного апельсинового сока, да поживее!

Программа производственной практики создает конвейер, по которому мы получаем отличных работников, но это достаточно долгий конвейер, и многие теряются по дороге. Обычно мы исходим из того, что для получения одного полноценного работника нам нужно пригласить двух практикантов, и если брать на практику тех, кому осталось год учиться, то между началом набора и временем, когда они впервые выйдут на работу

в качестве сотрудника компании, пройдет два года. В результате, каждое лето мы набираем столько практикантов, что в наших офисах почти не остается свободного места. Первые три года мы старались брать на практику только тех, кому осталось учиться один год, но стало понятно, что так мы теряем часть более юных способных студентов, поэтому была открыта программа для студентов любого года обучения. Я подумываю даже над тем, как нам привлечь старшеклассников, – к примеру, установить компьютеры, на которых после уроков они могли бы зарабатывать на колледж, – чтобы войти в контакт с новым поколением выдающихся программистов – даже если конвейер растянется до шести лет. Я далеко смотрю.

*Создание сообщества (*трудная задача)*

Идея заключается в том, чтобы создать большое сообщество сходно мыслящих толковых программистов, которые каким-то образом объединятся вокруг вашей компании, так, чтобы у вас появилась группа людей, к которой можно автоматически обратиться при появлении вакансии.

По правде говоря, именно так мы нашли многих замечательных сотрудников для Fog Creek: с помощью моего личного сайта «Joel on Software» (*joelonsoftware.com*). Главные статьи с этого сайта прочли, наверное, миллионы людей, большинство из которых занимается в том или ином качестве разработкой программного обеспечения. При такой огромной и отборной аудитории мне достаточно написать на главной странице, что нужен определенный работник, и обычно я получаю приличную пачку очень хороших резюме.

Эта категория помечена звездочкой, что означает «трудная задача», потому что такой совет похож на рекомендацию «чтобы выиграть конкурс красоты: а) станьте красивой, и б) примите участие в конкурсе». Дело в том, что я не вполне понимаю, почему мой сайт стал так популярен и отчего его посетители – лучшие программисты.

Увы, ничем больше в этом деле я вам помочь не могу. Дерек Поважек (Derek Powazek) написал на эту тему хорошую книгу «Design for Community: The Art of Connecting Real People in Virtual Places» («Проект сообщества: искусство соединения реальных людей в виртуальных местах»), New Riders, 2001. Многие компании пробовали различные стратегии блогинга, но, к сожалению, им не всегда удавалось собрать подходящую аудиторию, поэтому предупреждаю, что те средства, которые оказались действенными для нас, вам могут не подойти, и я не знаю, что вам посоветовать. Я открыл

сайт по занятости (*jobs.joelonsoftware.com*), где за 350 долларов вы можете дать объявление о работе, которое увидят посетители сайта «Joel on Software».

Рекомендация от сотрудника: скользкое дело

Стандартный совет по поиску выдающихся программистов – спросить у тех, кто уже работает у вас. При этом предполагается, что раз сами они толковые программисты, то и их знакомые – тоже толковые программисты.

Может, и так, но ведь они дружат и с теми, кто не очень силен как разработчик, и за всем этим кроется масса подводных камней, поэтому подбор кадров по рекомендации своих сотрудников я назвал бы одним из самых ненадежных способов.

Большой риск, конечно, связан с договорами об отказе от конкуренции. Если вы не задумывались об их значении, то вспомните случай с Crossgain, которой пришлось уволить четверть своих сотрудников, бывших работников Microsoft, когда та пригрозила судебными исками. Ни один программист, находясь в здравом уме, не должен подписывать договор об отказе от конкуренции, но большинство делают это, не веря, что его когда-либо пустят в ход, или потому, что не читают контракты, которые подписывают, или потому, что, приняв предложение о работе, уже перевезли семью с другого конца страны, а договор этот увидели только в первый день выхода на работу, и торговаться было уже поздновато. Поэтому они подписывают его, и это одна из самых гнусных уловок работодателей, потому что таким договорам *можно* дать законный ход, что часто и делается.

Договоры об отказе от конкуренции чреваты тем, что, принимая на работу по рекомендации своих сотрудников целую команду от одного работодателя, у которого, как вам сказали, есть классные программисты, вы серьезно рискуете.

Другая проблема связана с тем, что если у вас есть хоть какая-то система отсева кандидатов, то когда вы попросите своих сотрудников найти кого-нибудь среди их знакомых, они не предложат вам своих настоящих друзей. Никто не станет убеждать друга подать заявление о приеме на работу, если есть риск, что тому откажут и в результате пошатнется дружба.

Поскольку они не станут рассказывать вам о своих друзьях, а нанимать тех, с кем они раньше работали, рискованно, рассчитывать на обилие рекомендаций не приходится.

Но *настоящая* проблема со знакомыми сотрудников возникает у тех менеджеров, которые, не имея элементарных понятий об экономике, предлагают за такие рекомендации денежные премии. Такой подход не так уж редок. Расчет тут такой: если для поиска хорошего программиста воспользоваться услугами хедхантера или рекрутинговой компании, это обойдется примерно в 30–50 тысяч долларов. Если же мы заплатим своему работнику 5000 долларов за каждого, кого он приведет, или подарим дорогую машину за десяток рекомендаций, или расплатимся еще каким-то способом, то изрядно сэкономим. А 5000 сидящему на жаловании работнику кажутся немалыми деньгами. И впечатление такое, что все в выигрыше.

Беда в том, что внезапно вы обнаруживаете, как все завертелось, и ваши сотрудники тащат на интервью всех, кого только вспомнят, и у них есть сильный стимул для того, чтобы этих людей приняли, поэтому кандидатов натаскивают для прохождения собеседования, а в конференц-залах шепчутся с интервьюерами, и каждый из ваших сотрудников почему-то хочет, чтобы на работу взяли какого-нибудь соседа по общежитию, который вам совсем не нужен.

И этот метод не работает. Фирма ArsDigita прославилась тем, что купила и поставила на свою стоянку Ferrari, объявив, что получит машину тот, кто приведет 10 новых сотрудников. Никто даже не приблизился к такому результату, качество вновь принятых работников пошло на убыль, а компания прекратила свое существование, хотя, возможно, и не из-за Ferrari, который, как оказалось, был взят напрокат лишь для рекламного трюка.

Когда кто-то из сотрудников Fog Creek рекомендует нам «идеального работника», мы можем отменить только первичное телефонное собеседование. Обычно мы требуем, чтобы он прошел все собеседования, поскольку стремимся сохранить наши традиционно высокие стандарты.

Классификатор рабочих мест для программистов

На что ориентируются разработчики при поиске работы? Почему для них одно рабочее место привлекательнее другого? Как стать желанным работодателем для программистов? Читайте дальше!

ГЛАВА ТРЕТЬЯ

Классификатор мест работы для программистов

7 сентября 2006 года, четверг

К сожалению, вы можете дать объявления во всех нужных местах, организовать практику на самых лучших условиях, провести собеседование со всеми, с кем только пожелаете, но если выдающиеся программисты не захотят у вас работать, они не станут у вас работать. Данная глава служит своего рода определителем разработчиков: к чему они стремятся, что им нравится и не нравится на рабочем месте, и что нужно сделать, чтобы стать лучшим выбором для лучших программистов.

Личные кабинеты

В прошлом году я был в Йеле на конференции по вычислительным наукам. Один из выступавших, ветеран Силиконовой долины, бывший основателем или руководителем солидного списка стартапов с венчурным капиталом, показал присутствующим книгу «Peopleware» («Человеческий фактор») Тома Демарко (Tom DeMarco) и Тимоти Листера (Timothy Lister), Dorset House, 1999.

«Вам знакома эта книга, – сказал он. – Это библия управления софтверной компанией. Это самая важная из всех существующих книг об управлении софтверными компаниями».

Я согласен, «Peopleware» – выдающаяся книга. Одна из наиболее важных и спорных тем этой книги – утверждение о том, что высокую продуктивность программистов можно получить, только обеспечив им достаточ-

но места и тишину – по возможности, отдельные кабинеты. Авторы много рассуждают на эту тему.

Когда оратор закончил выступление, я подошел к нему.

– Согласен с вами насчет «Peopleware», – сказал я. – Скажите, пожалуйста, во всех ваших стартапах у программистов были отдельные кабинеты?

– Разумеется, нет, – отвечал он. – Капиталисты никогда не пошли бы на это.

Н-да.

– Но ведь это одна из важнейших тем этой книги, – сказал я.

– Да, но нужно оценивать свои силы. Венчурный капиталист просто решит, что вы пускаете его деньги на ветер.

В Силиконовой долине всегда считали, что надо запихнуть множество программистов в один большой зал, хотя я неоднократно приводил на своем сайте убедительные свидетельства того, что отдельные кабинеты значительно повышают их продуктивность. Кажется, я никого не убедил, потому что программисты *любят* общаться, даже в ущерб производительности, стало быть, моя задача оказалась неподъемной.

Мне даже приходилось слышать от программистов: «Да, мы работаем в кабинках, но так работают *все*, даже шеф!»

– Шеф? Ваш CEO работает в кабинке?

– Ну, у него кабинка, но, раз уж вы заговорили об этом, есть и большой конференц-зал, где он проводит все важные совещания...

Да-да. Типичное для Силиконовой долины явление: CEO демонстративно работает в кабинке, как простой смертный, хотя есть вот эта комната для совещаний, которую он обычно приватизирует («только на случай конфиденциальных переговоров», хотя, как ни идешь мимо, – вечно он там сидит один и болтает по телефону с приятелем из гольф-клуба, при этом его штиблеты Cole Haan парят над столом для совещаний).

Тем не менее сейчас мне не хочется опять пускаться в рассуждения о том, почему благодаря отдельным кабинетам программисты работают продуктивнее, или почему когда программист надевает противошумные наушники, качество его работы, как оказалось, падает, и почему в действительности устройство отдельных кабинетов для программистов не требует таких уж больших дополнительных расходов. Все это я уже говорил. Сегодня речь пойдет о найме работников и роли отдельных кабинетов в этом деле.

Что бы вы ни думали относительно производительности и равных условий работы, бесспорны две вещи:

1. Отдельные кабинеты котируются выше.
2. Кабинки и прочие виды совместно используемого пространства могут вызывать социальный дискомфорт.

Из этих двух фактов я делаю вывод, что программисты предпочитают ту работу, где им предоставляется отдельный кабинет, особенно с дверью и хорошим видом из окна.

Сложность в том, что некоторые решения, способные облегчить вам задачу привлечения новых сотрудников, не всегда зависят именно от вас. Даже у главы фирмы и ее соучредителей может не быть отдельных кабинетов, если они зависят от венчурного капиталиста. В большинстве компаний переезд на новое место или реорганизация пространства происходят не чаще, чем раз в 5–10 лет. Небольшие начинающие компании не всегда могут позволить себе устроить отдельные кабинеты. Поэтому, как я убедился, отговорок находится столько, что практически нигде, кроме наиболее передовых компаний, вы не найдете отдельных кабинетов для программистов, и даже там решение о том, куда переезжать и где будут работать люди, принимается раз в 10 лет комиссией в составе секретаря офис-менеджера и скромного сотрудника архитектурной фирмы, которому в колледже напели о том, что открытые пространства – это открытые компании или что-то в этом роде – практически без какого-либо участия самих разработчиков.

Это довольно позорное явление, и я буду с ним бороться, а между тем проблема отдельных кабинетов *решаема*: мы, как правило, обеспечивали отдельными кабинетами всех своих программистов, работающих на полной ставке, даже в Нью-Йорке, где арендная плата одна из самых высоких в мире, и, безусловно, благодаря этому людям гораздо приятнее работать в Fog Creek, так что если все против – *так и быть*, пусть это конкурентное преимущество остается за мной.

Физическое рабочее пространство

Физическое рабочее пространство не ограничивается отдельным кабинетом. Придя в компанию на собеседование, кандидат осматривается, замечая, как работают люди, и пытаясь представить себя в этой среде. Если офис выглядит приятно: светлый, расположен в хорошем районе, все новое и чистое – у кандидата возникают приятные мысли. Тесный офис с затоптанными полами и некрашеными стенами, на которых развешаны

плакаты с изображением команды гребцов и надписью крупным шрифтом МЫ ОДНА КОМАНДА, порождает мысли в духе Дилберта.

Многие технические специалисты удивительно мало внимания обращают на общее состояние своего офиса. Даже те, кто ценит офисный комфорт, могут не замечать отдельные недостатки собственного офиса, поскольку вполне к ним привыкли.

Поставьте себя на место кандидата, который к вам пришел, и ответьте честно:

- Что он подумает о местонахождении фирмы? Как Буффало выглядит в сравнении, скажем, с Остином? Захочется ли кому-то переехать в Детройт? Если вы располагаетесь в Буффало или Детройте – не попробовать ли хотя бы проводить собеседования в сентябре?
- Что видит тот, кто приходит к вам в офис? Восхитительное чистое помещение? Фойе с пальмами и фонтаном – или нечто похожее на бесплатную зубную клинику в трущобах, с засохшим букетом и старыми номерами «Newsweek»?
- Как выглядит ваше рабочее место? Все новенькое и блестит? Или у вас все еще висит гигантский пожелтевший плакат TEAM BANANA, распечатанный матричным принтером на перфорированной бумаге в те годы, когда еще существовали такие понятия, как перфорированная бумага и матричные принтеры?
- Какие у вас рабочие столы? Что на столах у программистов – несколько больших плоских экранов или один электронно-лучевой монитор? Какие стулья – Aeron или Staples Special?

Пара слов о знаменитых офисных креслах Aeron, которые выпускает Herman Miller. Стоят они около 900 долларов. Это на 800 долларов дороже дешевого офисного кресла, купленного в Office Depot или Staples.

Они *гораздо* удобнее дешевых кресел. Если правильно выбрать размер такого кресла и отрегулировать его, можно легко просидеть на нем целый день. Спинка и сиденье сделаны из сетки и пропускают воздух, поэтому не потеешь. Превосходная эргономичность, особенно у последних моделей с опорой для поясницы.

Они долговечнее дешевых кресел. Они у нас уже шесть лет и выглядят как новенькие – предлагаю всем сравнить те, которые мы купили в 2000 году и три месяца назад. Они легко продержатся десяток лет. Дешевые кресла начинают разваливаться буквально через месяцы. За то время, что прослужит Aeron, кресла по 100 долларов придется сменить как минимум четырежды.

Стало быть, Aeron обойдется вам всего в дополнительные 500 долларов на десять лет, или по 50 долларов в год. Доллар в неделю на программиста.

Рулон хорошей туалетной бумаги стоит примерно доллар. На каждого вашего программиста нужен примерно рулон в неделю.

Следовательно, переход на кресла Aeron обойдется буквально во столько же, сколько фирма тратит на *туалетную бумагу*, и, поверьте, если вы придете в бюджетную комиссию с заявкой на туалетную бумагу, вам строго укажут, чтобы вы не отвлекали людей пустяками.

К сожалению, за креслом Aeron закрепилась репутация неэкономности, особенно для начинающих компаний. Оно почему-то стало символом бездарно растраченных во время дотком-бума капиталов, что несправедливо, поскольку с учетом долговечности оно не так уж дорого. В самом деле, ведь в нем проводишь по восемь часов в день, так что даже самые навороченные модели с поддержкой поясницы и этим чертовым *хвостовым оперением* кажутся такими дешевыми, что, покупая их, практически *получаешь* прибыль.

Игрушки

Та же логика применима и к другим игрушкам, которыми пользуются разработчики. Нет никаких причин не купить для них лучшие модели компьютеров, хотя бы пару больших (21 дюйм) ЖК-дисплеев (или один 30-дюймовый), и предоставить свободу заказывать на *Amazon.com* любую нужную им техническую литературу. Очевидно, что это повышает их продуктивность. Но сейчас, когда мы обсуждаем, как набирать работников, важнее как раз то, что такие вещи производят впечатление на кандидатов, особенно в условиях, когда большинство компаний относится к программистам как к расходному материалу или как к машинисткам – «Ну зачем им большой монитор, и чем плоха 15-дюймовая трубка? Вот когда *я* был молодым...» и т. д.

Общественная жизнь программистов

Разработчики программ не столь уж сильно отличаются от обычных людей. Разумеется, мне известно распространенное представление о программистах как о гиках с синдромом Аспергера, которым совершенно чуждо межличностное общение, но это совсем не так. Даже гиков с синдромом Асперегера интересуют социальные аспекты их рабочего места, рассмотренные ниже.

Как организация относится к программистам?

Как к искусным мастерам или как к простым машинисткам? Есть ли в руководстве компании инженеры или бывшие программисты? Если разработчики летят на конференцию, оплачивают ли им первый класс? (Можете считать это пустой тратой денег. Звезды летают первым классом. Привыкайте к этому.) Если они прилетают на собеседование, присылают ли за ними лимузин или они должны самостоятельно добираться до офиса? При всех прочих равных условиях разработчики предпочтут организацию, которая носится с ними как со звездами. Если ваш президент – ворчливый выходец из отдела продаж, которому непонятно, почему эти примадонны от программирования постоянно требуют себе то подушечки под запястья, то большие мониторы, то удобные кресла – кем они себя возомнили?! – вашей компании, скорее всего, требуется корректировка поведения. Выдающиеся разработчики не станут работать у вас, если вы не будете относиться к ним с уважением.

Что представляют собой их коллеги?

Большое внимание программисты, прибывающие на собеседование, обращают на то, с какими людьми они встречаются в этот день. Приятно ли с ними общаться? Еще важнее: умны ли они? Когда-то я проходил летнюю практику в Bellcore, дочерней компании Bell Labs, и все, с кем я встречался, повторяли одно и то же: самое замечательное в Bellcore – это люди, с которыми там работаешь.

Поэтому если у вас есть хмурые программисты, от которых вам никак не избавиться, хотя бы не привлекайте их к участию в собеседованиях, а веселых и общительных включите обязательно, если такие есть. Имейте в виду, из-за того, что кандидат встречал в вашей компании одни лишь угрюмые лица, он не вспомнит о ней с теплом, когда вернется домой и будет решать, где ему работать.

Кстати, сначала мы в Fog Creek руководствовались при приеме программистов заимствованным у Microsoft правилом «Smart, and Gets Things Done» (умный и доводит дело до конца). Но еще не приступив к делу, поняли, что нужно добавить еще одно условие: «Not a jerk[1]» (порядочный). Припоминаю, что для получения работы в Microsoft это требование не было

[1] У этого слова много сленговых значений, в том числе «наглец», «негодяй» и «придурок». – *Прим. перев.*

обязательным; хотя, уверен, там стали бы разглагольствовать о том, как важны добрые отношения между людьми, но, тем не менее, никому не отказали бы в приеме на работу лишь по той причине, что он наглец. На самом деле, иногда похоже на то, что отсутствие порядочности – нечто вроде пропуска в высшее руководство. С точки зрения бизнеса, это не такой уж недостаток, а с точки зрения приема новых работников – да, потому что кто же захочет работать в компании, где терпят наглецов?

Независимость и самостоятельность

Когда в 1999 году я уходил из Juno, – перед тем, как создать Fog Creek Software, – в отделе кадров провели со мной стандартное собеседование в связи с увольнением, и тогда я поддался соблазну рассказать кадровику об ошибках руководства компании – не сомневаясь, что это лишь повредит мне, я, тем не менее, высказал все, упирая на свойственный Juno стиль «молниеносного» управления. Обычно менеджеры не мешали людям спокойно делать свое дело, но временами они вдруг влезали в самые мелкие детали, требуя сделать что-то в точности так, как они хотят, а потом, не дожидаясь нелепых результатов выполнения своих указаний, перескакивали к следующей задаче микроуправления. Помню, например, особо досадный эпизод, когда в течение двух-трех дней все, от непосредственного начальника до главы компании, указывали мне, как должны вводиться даты в форме для регистрации на сайте компании. Они не были специалистами по пользовательскому интерфейсу, но и не стремились обсудить со мной проблему, чтобы понять, почему в данном случае прав я, и все было бесполезно: руководство не отступалось от проблемы, не внимая при этом моим доводам.

В принципе, собираясь нанимать толковых работников, вы должны быть готовы дать им возможность проявить свое мастерство в работе. Менеджеры могут давать советы, что приветствуется, но они должны тщательно следить за тем, чтобы их «советы» не воспринимались как приказы, потому что в любом техническом вопросе руководство, как правило, разбирается хуже, чем рядовые сотрудники, особенно если, как я сказал, вы наняли толковых людей.

Разработчики хотят наниматься в качестве профессионалов, чтобы, признавая их мастерство, им позволяли принимать решения в той области, где они являются специалистами.

Никакой политики

Фактически, политика появляется всюду, где собирается вместе больше двух человек. Это естественно. Говоря «никакой политики», я имею в виду «никакой нездоровой политики». У программистов очень острое чувство справедливости. Код либо работает, либо нет. Нет смысла спорить, есть ошибка или нет, поскольку это определяется путем тестирования кода. Мир программирования очень справедлив и строго упорядочен, и многие приходят в программирование в первую очередь потому, что хотят пребывать в справедливой и добропорядочной среде, где ценятся заслуги, а в споре побеждает тот, кто действительно *прав*.

И такую среду вы должны создать, если хотите заполучить программистов. Когда программист жалуется на «политику», совершенно ясно, что он имеет в виду ситуацию, в которой личные предпочтения доминируют над технической целесообразностью. Ничто так не возмущает разработчика, как указание использовать определенный язык программирования (не самый подходящий для решаемой задачи) только потому, что этот язык нравится начальнику. Ничто так не бесит, как карьерное продвижение благодаря умелым интригам, а не личным достоинствам. Ничто так не удручает программиста, как необходимость реализовать технически слабое решение, на котором настаивает кто-то вышестоящий или обладающий личными связями с руководством.

Ничто не приносит такого удовлетворения, как победа в споре благодаря техническим аргументам, особенно если политическая обстановка угрожала поражением. Когда я начал работать в Microsoft, основной разработкой там был неудачный проект под названием MacroMan, посвященный созданию графического языка макропрограммирования. Этот язык стал бы сплошным разочарованием для профессиональных программистов, потому что его графическая природа не позволяла создавать циклы и условные операторы, да и непрограммистам он бы не очень помог, потому что они, по моему мнению, не привыкли мыслить алгоритмически и вообще ничего не поймут в MacroMan. Когда я поведал об этом своему начальнику, он сказал мне: «*Этот* паровоз тебе не остановить. И не пытайся». Но я продолжал доказывать, и доказывать, и доказывать – а ведь я был вчерашний студент, и едва ли у кого-то в Microsoft было меньше личных связей, чем у меня тогда, – в конечном итоге к моим аргументам прислушались, и проект MacroMan был закрыт. Неважно, кем я был, – важно, что я был прав. Такая неполитическая обстановка очень радует программистов.

В конечном счете, забота о социальной динамике вашей организации имеет решающее значение для создания здоровой, комфортной рабочей обстановки, которая удержит ваших программистов и привлечет новых.

Над чем я работаю?

Иногда можно привлечь новых разработчиков, предложив им интересную задачу. Но выбор задачи не всегда в ваших силах: если вы разрабатываете программное обеспечение для производства песка и гравия, то не сможете выдать себя за крутой новый веб-проект, чтобы привлечь программистов.

Еще программистам может понравиться, если им предлагают работать над достаточно простыми и популярными вещами, о которых можно поговорить с тетушкой Ирмой, навестив ее в День благодарения. Тетушка Ирма, вестимо, физик-ядерщик и не очень-то понимает в программировании на Ruby для песчаных карьеров.

Наконец, для многих сотрудников важна общественная значимость компании, где они работают. Работа в компании, поддерживающей социальную сеть или блог, помогает людям общаться и не загрязняет окружающую среду, поэтому пользуется популярностью, тогда как работа в военной промышленности или в сомнительных с этической точки зрения компаниях, известных финансовыми махинациями, менее популярна.

К сожалению, я не уверен, что смог бы подсказать обычному менеджеру, как использовать данный фактор для привлечения новых работников. Можно попытаться изменить номенклатуру продукции, включив в нее что-то «крутое», но ваши возможности здесь ограничены. Тем не менее вот ряд приемов, которые применяют некоторые компании, пытаясь действовать в этом направлении:

Позвольте лучшему из кандидатов самому выбрать проект

Долгие годы в Oracle Corporation существовала программа MAP (Multiple Alternatives Program – программа нескольких альтернатив). Она предлагалась выпускникам колледжей, которые считались лучшими кандидатами в каждом классе. Идея заключалась в том, что они могли прийти в Oracle, потолкаться там неделю-другую, посещая все группы, где есть свободные места, а потом выбрать из них ту, где им больше понравилось.

Думаю, идея была хорошей, но только в Oracle знают, оказалась ли она успешной.

Применяйте крутые новые технологии не только по необходимости

Крупные инвестиционные банки Нью-Йорка считаются достаточно суровыми местами для программистов. Условия работы ужасны: долгий рабочий день, шум, тираническое начальство; программисты здесь явно граждане третьего сорта, тогда как элитой считаются лопающиеся от тестостерона человекообразные, которые продают финансовые инструменты, получая премии по 30 000 000 долларов и обжираясь чизбургерами (за которыми могут послать оказавшегося поблизости программиста). Таков стереотип, и удержать лучших программистов инвестиционные банки могут двумя способами: платить им кучу денег и дать возможность почти бесконечно переписывать все заново на тех модных языках программирования, которые им понравятся. Хочешь переписать все приложение торгов на Lisp? Да на чем угодно. Только принеси мне чертов чизбургер.

Некоторым программистам совершенно безразлично, на каком языке писать, но большинство с восторгом отнесется к возможности поработать с интересными новыми технологиями. Сегодня это может быть Python или Ruby on Rails; три года назад это был C#, а еще раньше – Java.

Я вовсе не отговариваю вас использовать для каждой задачи лучший инструмент и не предлагаю каждые два года переписывать приложение на самом модном языке, но если вы изыщете возможность дать разработчикам поработать с новыми языками, средами и технологиями, они будут счастливы. Даже если вы не рискнете переписать свое базовое приложение – что мешает запустить учебный проект и переписать внутрифирменные инструменты или менее критичные приложения на каком-нибудь замечательном новом языке?

Могу ли я отождествить себя с компанией?

Большинству программистов не свойственно браться за любую работу, лишь бы она позволяла оплачивать жилье. Они не ищут «работу без сверхурочных»: они хотят, чтобы их работа имела смысл. Они хотят отождествлять себя со своей компанией. Молодых программистов в особенности привлекают идеологические компании. Множество компаний име-

ют какое-то отношение к open source или к движению за свободное программное обеспечение (это не одно и то же), и это может оказаться привлекательным для идеалистически настроенных программистов. Другие компании поддерживают какие-нибудь общественные начинания или производят продукт, который можно воспринимать или изображать как приносящий пользу обществу.

Набирая сотрудников для своей компании, вы должны определить ее идеалистические аспекты и постараться ознакомить с ними кандидатов.

Некоторые компании даже пытаются создать собственное идеологическое движение. Начинающая чикагская компания 37signals активно рекламирует себя как сторонницу идеи простоты: простые и удобные приложения вроде Backpack, простая и удобная программная среда Ruby on Rails.

Для 37signals простота – это «-изм», практически международное политическое движение. Простота – это не так просто, о нет, это лето, прекрасная музыка, мир, справедливость, счастье и прелестные девушки с цветами в волосах. Создатель Rails Д. Х. Хэнссон (David Heinemeier Hansson) называет это идеей «красоты, счастья и движущей силы. Гордость и наслаждение своей работой и своими инструментами. Это не каприз, это тенденция. Это идея, побуждающая разработчика искренне и без смущения говорить о страсти и энтузиазме, потому что он действительно любит то, что делает» (*www.loudthinking.com/arc/2006_08.html*). Вознесение среды программирования веб-приложений на уровень «красоты, счастья и движущей силы» может показаться высокомерием, но это весьма привлекательно и явно выделяет компанию среди других. Идея Ruby on Rails, преподнесенная как Счастье, практически гарантирует, что хотя бы некоторые из разработчиков постараются найти работу, связанную с Ruby on Rails.

Но 37signals – новички в таких приемах отождествления. Они и *в подметки* не годятся Apple Computer, которая в 1984 году с помощью одного лишь рекламного ролика во время розыгрыша Суперкубка умудрилась утвердить и сохранить *до сего дня* свое положение как относящейся к контркультуре силы, ведущей борьбу за свободу против диктатуры, за вольность против угнетения, за цветную картину против черно-белой, за красивых женщин в ярко-красных шортах против безликих мужчин в костюмах. Боюсь, что последствия до смешного напоминают Оруэлла: гигантские корпорации манипулируют своим имиджем в обществе при помощи абсурдных средств (если это компьютерная компания – то причем тут борьба с диктатурой?) и успешно создают культуру отождествления, при которой покупатели во всем мире чувствуют, что приобретают не просто компью-

тер, а *долю в общественном движении*. Ну да, покупая iPod, вы, конечно, поддерживаете борьбу Ганди с британским колониализмом. Каждая покупка MacBook – это выступление против диктатуры и голода!

Как бы то ни было. Сделайте глубокий вдох... Цель этого раздела – побудить вас задуматься о том, что представляет собой ваша компания, как она выглядит в данное время со стороны и какой могла бы быть. Ваш корпоративный бренд так же важен для набора сотрудников, как и для маркетинга.

Что не волнует программистов

Их фактически не волнуют деньги – до того момента, когда вы испортите им жизнь чем-то другим. Если вдруг пошли жалобы на зарплату, чего раньше не бывало, обычно это признак того, что людям перестала нравиться их работа. Если кандидаты на работу требуют непомерных окладов, отказываясь снизить свои требования, не исключено, что при этом они думают: меня будет тошнить от этой работы, но я потерплю, если они мне хорошо заплатят.

Это не значит, что можно недоплачивать своим работникам, потому что они стремятся к справедливости и придут в ярость, узнав, что разным людям платят разное жалование за одинаковую работу или что в вашей фирме всем платят на 20% меньше, чем в точно такой же фирме за углом, и тогда деньги превратятся в серьезную проблему. Вы должны платить согласно общепринятым расценкам, но при всем при этом среди факторов, которые оценивают программисты, выбирая место работы, зарплата, если она достаточно справедлива, занимает далеко не первое место, и предложение высокой зарплаты оказывается удивительно неэффективным средством компенсации таких проблем, как 15-дюймовые мониторы для программистов, постоянное хамство агентов по сбыту и то, что работа связана с изготовлением ядерного оружия из бельков.

ГЛАВА ЧЕТВЕРТАЯ

Три способа управления (введение)

7 августа 2006 года, понедельник

Если вы хотите руководить командой, армией или страной, главная ваша задача – сделать так, чтобы все двигались в одном направлении (на самом деле, это значит «заставить людей делать то, что вам нужно», просто в более вежливой форме).

Имейте в виду следующее. Если в вашей команде есть еще хоть один человек, значит, у вас есть разные люди с разными планами. У них с вами разные цели. Если вы организовали стартап-компанию, то, вероятно, хотите быстро заработать много денег и рано уйти на отдых и двадцать лет посещать конференции женщин-блогеров. Поэтому вы можете постоянно разъезжать по Сэнд Хилл Роуд и искать венчурных капиталистов, которые купят вашу компанию и скинут ее Yahoo!. Но программистку Дженис, вашу сотрудницу, не интересует, купит ли Yahoo! компанию, потому что она ничего при этом не заработает. Ее интересует, как писать код на крутом новом языке программирования, потому что она любит изучать все новое. Тем временем вашим финансовым директором завладела мысль выбраться из кабинки, которую он делит с вашим системным администратором Трекки Монстром, поэтому он разрабатывает новое предложение по бюджету, в котором доказывает, как много можно сэкономить, переехав в большой офис, расположенный – какое совпадение! – всего в двух минутах от его дома.

Разумеется, вопрос о том, как заставить всех двигаться *в нужную вам* сторону (или хотя бы *в одну и ту же* сторону), возникает не только у стартапов. Это та же фундаментальная проблема, с которой сталкивается политический деятель после победы на выборах, перед которыми он обещал очистить правительство от бездельников, коррупционеров и мошенников. Мэр хочет, чтобы городское собрание одобрило новый строительный проект. Строительные инспекторы городского собрания хотят по-прежнему получать взятки, к которым привыкли.

Та же проблема и у военных. Генерал хочет, чтобы его солдаты стреляли по врагу, а солдат хочет спрятаться за скалу и чтобы за него стрелял кто-то другой.

Вот три основных метода управления, которыми можно воспользоваться:

- Командно-административный
- Экономика 101
- Отождествление

Несомненно, вы обнаружите и другие методы (экзотический метод «Дьявол носит Прада», метод джихада, метод харизматического культа и метод «то один, то другой метод»), но в следующих трех главах, посвященных этим трем популярным методам, исследуются их сильные и слабые стороны.

Далее: командно-административный метод управления.

ГЛАВА ПЯТАЯ

Командно-административный метод управления

8 августа 2006 года, вторник

Солдат должен бояться своего командира больше, чем любой грозящей ему опасности... Солдат никогда на станет добровольно сражаться с этими опасностями: он сделает это только из страха.

Фридрих Великий

Командно-административная система управления основана на методах, применяемых в военном деле. Суть идеи в том, что человек должен выполнять приказ, а если он этого не желает, нужно орать на него, пока не подчинится, если и это не помогло, то посадить его на какое-то время на гауптвахту, а если и это ничему его не научило, то отправить его на подводную лодку чистить лук нос к носу с парнем из деревни, имеющим лишь отдаленное представление о зубной щетке.

Разнообразие методов велико. Некоторые идеи можно почерпнуть из фильмов «Блюз Билокси» и «Офицер и джентльмен».

Некоторые менеджеры пользуются этим методом, потому что их действительно приучили к нему в армии. Другие выросли в авторитарных семьях или государствах, и им кажется, что это естественный способ добиться согласия. Третьи просто не могут придумать ничего лучше. Если это срабатывает в армии – должно сработать и в начинающей интернет-компании!

Однако применение этого метода в команде, занимающейся высокими технологиями, имеет три недостатка.

Во-первых, этот метод не нравится людям, особенно таким умникам, как разработчики программного обеспечения, которые действительно довольно умны и считают, что знают больше других, причем не без оснований, потому что так оно и есть, и поэтому им крайне неприятно, если кто-то приказывает им сделать это только «потому, что так надо». Но это недостаточно веская причина, чтобы отбросить данный метод... Попытаемся найти более рациональные. У команд в высокотехнологичных областях множество задач, но вряд ли главная из них – сделать всех счастливыми.

Более существенным недостатком командно-административного метода является то, что у менеджеров просто нет времени, чтобы осуществлять руководство на таком низком уровне, потому что для этого требуется слишком много менеджеров. В армии можно отдать приказ сразу большой группе людей, потому что часто все они выполняют одни и те же действия. «Приступить к чистке оружия!» – командуете вы взводу из 28 человек, после чего идете вздремнуть или выпить холодного чаю на веранде офицерского клуба. В команде разработчиков каждый работает над своей отдельной задачей, поэтому попытки микроуправления делаются в стиле *молниеносного микроуправления*. То есть в приливе активности вы даете разработчику самые детальные указания, а потом на пару недель исчезаете из его жизни, применяя микроуправление к другим разработчикам. Проблема в том, что при таком стиле вы не можете задержаться на одном месте, чтобы проверить, правильно ли ваше решение и не следует ли его скорректировать. В результате вы только время от времени будете сталкивать своих несчастных программистов с накатанной колеи, после чего им придется неделю собирать свои вагончики и ставить их обратно на рельсы, чтобы двигаться дальше, чувствуя себя при этом несколько потрепанными.

Третий недостаток связан с тем, что в высокотехнологичной компании отдельный сотрудник всегда знает больше «руководителя», поэтому у него действительно больше оснований принять лучшее решение. Когда босс забредает в кабинет, где два разработчика целый час спорят, как лучше сжать графический файл, то из всех трех *наименьшей* информацией располагает босс, поэтому он последний, кому следовало бы дать право принятия *технического* решения. Я помню, что когда Майк Мэйплс был моим «большим начальником», являясь руководителем Microsoft Applications, он твердо отказывался принимать чью-то сторону в технических проблемах. В конечном счете, все усвоили, что бесполезно идти к нему за вердиктом.

Пришлось обсуждать проблемы с технической стороны и выбирать то решение, автор которого был самым красноречивым... ну, то есть выбирать самое рациональное решение.

Если командно-административный метод столь плох, почему же им пользуются военные?

Мне разъяснили это в сержантской школе. В 1986 году я служил в израильских воздушно-десантных войсках. Сейчас мне кажется, что я был, пожалуй, худшим десантником из всех, кто когда-либо там служил.

Было несколько постоянно действующих приказов-инструкций для солдат. Инструкция номер один: попал на минное поле – *замри*. Разумно, правда? Время от времени инструктор выкрикивал: «Мины!» – и все должны были замереть, так что вскоре это сделалось привычкой.

Инструкция номер два: если тебя атаковали, *беги навстречу атакующему, ведя огонь*. Огонь заставит его уйти в укрытие, поэтому он не сможет стрелять в тебя. Если бежать навстречу противнику, он приближается, так что в него легче прицелиться и попасть. Тоже разумная инструкция.

А теперь вопрос для собеседования: что делать, если ты оказался на минном поле и тебя атаковали?

Это не выдуманная ситуация – это такая досадная вещь, как засада.

Оказывается, правильный ответ: не обращая внимания на минное поле, бежать навстречу атакующему, ведя огонь.

Потому что если оставаться на месте, то всех перестреляют одного за другим, а на минах подорвутся не все, поэтому такое поведение целесообразнее.

Проблема в том, что ни один разумный солдат в таких обстоятельствах не захочет атаковать. У каждого возникает сильнейшее желание схитрить: я пока замру, а атакуют пусть другие, более мужественные. Своего рода Дилемма заключенного.

В ситуациях на грани жизни и смерти командир должен быть уверен, что солдаты выполнят его приказ, даже если им придется погибнуть. Значит, от солдат требуется та степень повиновения, которой нет никакой необходимости добиваться, скажем, компании-производителю программного обеспечения.

Иными словами, командно-административный метод применяется в армии, потому что это единственный способ заставить 18-летних парней бежать по минному полю, а не потому, что это лучший метод во всех ситуациях.

В частности, если применять армейские порядки к команде разработчиков в условиях, когда хорошие программисты всегда могут найти себе другое место, им это быстро надоест, и от вас все разбегутся.

Далее: Метод управления «Экономика 101».

ГЛАВА ШЕСТАЯ

Метод управления «Экономика 101»

9 августа 2006 года, среда

Анекдот: Россия, XIX век. К бедному местечковому еврею подъезжает на коне казак и спрашивает:

– Чем курицу кормишь?

– Хлебными крошками, – отвечает еврей.

– Да как ты смеешь так плохо кормить эту прекрасную русскую курицу? – кричит казак и хлещет еврея плеткой.

На следующий день казак приезжает снова и спрашивает:

– А теперь чем курицу кормишь?

– Обедом из трех блюд. На первое молодая спаржа, потом черная икра и, наконец, на десерт французский шоколад и взбитые сливки.

– Идиот! – кричит казак и снова хлещет еврея плеткой. – Как ты смеешь переводить такую замечательную еду на какую-то курицу!

На третий день казак снова спрашивает:

– Чем курицу кормишь?

– Ничем, – взмолился еврей, – я даю ей копейку, и она покупает себе все, что захочет.

(Можно смеяться)

(Не хотите?)

(Тыц-тыц-тыц)

(Не слышу смеха)

(Ну и ладно)

Название «Экономика 101» я взял для иронии. Дело в том, что на большинстве факультетов американских колледжей есть курс с номером 101, представляющий собой введение в какую-либо дисциплину. Управление в стиле «Экономика 101» практикуют те, кто достаточно наслышан об экономике, чтобы представлять собой угрозу обществу.

Менеджер стиля «Экономика 101» считает, что мотивацией любого человека служат деньги и что заставлять людей делать то, что нужно, лучше всего с помощью денежных вознаграждений и вычетов.

Например, AOL может доплачивать служащим своего центра обработки вызовов за каждого клиента, которого те уговорят *не* прекращать свою подписку.

Софтверная фирма может выплачивать премии программистам, у которых обнаруживается меньше всего ошибок.

Это примерно столь же успешный метод, как выдача курам денег, чтобы они питались самостоятельно.

При этом происходит замена внутренней мотивации внешней, что влечет серьезные проблемы.

Внутренняя мотивация – это ваше естественное стремление работать хорошо. Обычно поначалу у человека есть сильная внутренняя мотивация. Он хочет делать свое дело хорошо. Он *хочет* помочь другим убедиться в том, что продолжать платить AOL 24 доллара в месяц – в их собственных интересах. Он *хочет* писать код, в котором мало ошибок.

Внешняя мотивация приходит снаружи, например, когда вам платят за что-то особенное.

Внутренняя мотивация гораздо сильнее внешней. Люди гораздо усерднее занимаются тем, что им *действительно нравится*. Тут и спорить не о чем.

Но когда вы предлагаете людям деньги за то, что они и так хотят делать, возникает нечто под названием «эффекта сверхоправдания». «Я должен писать код без ошибок, потому что хочу получать за это деньги», – думает программист, и внешняя мотивация *вытесняет* в нем внутреннюю. Поскольку внешняя мотивация гораздо менее продуктивна, в итоге вы добиваетесь фактического *ослабления* желания хорошо выполнять работу. Если вы прекратите выплачивать премию, или программист решит, что деньги его не очень интересуют, он перестанет следить за тем, чтобы в коде не было ошибок.

Другая проблема метода «Экономика 101» связана со склонностью людей искать локальные максимумы. Человек непременно отыщет способ оптимизировать то, за что вы ему доплачиваете, и вы не получите то, чего от него добивались.

Например, ваш специалист по удержанию клиентов, стремясь получить премию за сохраненного клиента, доведет его до такого состояния, что на первой странице «New York Times» появится статья об отвратительном поведении вашей службы работы с клиентами. Его действия максимизируют то, за что вы ему платите (удержание клиентов), но не то, в чем состоит ваш главный интерес (прибыль). А когда вы позже попытаетесь связать его интересы с пользой для компании, скажем, выдав ему тринадцать акций, то поймете, что на самом деле он на это никак не влияет, так что все это пустая трата времени.

Применяя управление по методу «Экономика 101», вы вызываете у разработчиков желание перехитрить систему.

Допустим, вы решили выплатить премию разработчику, у которого меньше всего ошибок. Теперь при каждом сообщении тестера о найденной им ошибке будет возникать спор, в котором разработчик, как правило, сможет доказать тестеру, что это вовсе не ошибка. Или же тестер согласится «неформально» сообщить об ошибке разработчику, не регистрируя ошибку в журнале. В итоге никто не будет пользоваться системой слежения за ошибками. Счетчик ошибок падает, а количество ошибок остается прежним.

Разработчики – ушлые в этом смысле ребята. Что бы вы ни пытались измерить, они найдут способ максимизировать это, и вы никогда получите точных результатов.

Роберт Д. Остин (Robert D. Austin) в своей книге «Measuring and Managing Performance in Organizations» («Производительность организаций: измерение и управление») отмечает, что новые метрики вводятся в два этапа. Сначала вы действительно получаете то, что хотите, потому что никто еще не догадался, как обмануть систему. На втором этапе ваше положение оказывается *хуже*, чем до нововведения, потому что все уже знают, как максимизировать то, что вы измеряете, пусть даже с ущербом для компании.

К сожалению, менеджеры стиля «Экономика 101» полагают, что здесь поможет правильная корректировка метрики. Но Роберт Д. Остин утверждает, что это *невозможно*. Как бы вы ни пытались настроить метрики, чтобы они отражали действительное положение дел, результат всегда отрицательный.

Но главная проблема управления по методу «Экономика 101» в том, что это вовсе не управление – скорее, это уход от управления. Намеренный отказ от поиска решения, улучшающего положение. Это признак того, что руководство просто не знает, как научить людей работать лучше, и поэтому заставляет всех участников системы искать свои способы.

Вместо того чтобы научить программистов писать надежный код, вы просто снимаете с себя ответственность и платите им, если у них это получится. Теперь каждый разработчик ищет решение на свой страх и риск.

В достаточно обыденных занятиях, таких, как работа за прилавком Starbucks или ответы на звонки клиентов AOL, очень маловероятно, что рядовой работник сам придумает, как можно успешнее справляться с его работой. Зайдите в любую кофейню и закажите «маленький с молоком и сахаром, погорячее» – и вам придется повторить свой заказ несколько раз: сначала тому, кто варит кофе, потом еще раз ему же, потому что он забыл, что вы заказали, и наконец кассиру, чтобы он мог посчитать стоимость заказа. Никто не научил их работать лучше. Нигде не знают, как это делается, кроме кофеен Starbucks, работник которых знает всю номенклатуру наизусть, записывает на чашке и выкрикивает заказ, в результате клиенту не приходится повторять больше одного раза, что он будет пить. Система, изобретенная руководством Starbucks, отлично работает, но работники других сетей никогда не дойдут до этого сами.

Сотрудники нашей службы поддержки клиентов большую часть дня проводят в разговорах с клиентами. У них нет ни времени, ни склонности, ни достаточной подготовки, чтобы придумывать более эффективные методы работы. Никому из группы удержания клиентов не под силу, отшивая кое-кого из блогеров, одновременно заниматься статистикой и измерениями, чтобы оценить самый эффективный способ удержания клиентов. Их это не очень интересует, они недостаточно сообразительны для этого, у них мало информации или они просто слишком заняты своей повседневной работой.

Вы менеджер – вам и придумывать систему. За это вам и платят «большие бабки».

Если в детстве вы слишком увлекались книгами Эйн Рэнд (Ayn Rand) или прослушали только первый семестр курса экономики – где еще не объясняют, что полезность измеряется не долларами, – то вам может показаться, что упрощенная схема премирования и оплаты в зависимости от производительности или качества работы (Pay For Performance) – непло-

хой метод управления. Но он не действует. Лучше займитесь управлением по-настоящему и перестаньте кормить своих кур копейками.

«Джоэл, – возмутитесь вы, – в предыдущей главе ты доказывал, что решения должны принимать разработчики, а сейчас говоришь, что решения должны принимать менеджеры. Что за дела?»

Положим, это не совсем так. В предыдущей главе я объяснял, что разработчики, то есть листья дерева, знают больше других; микроуправление, или командный метод, обычно не приводит к оптимальным результатам. Здесь же я объясняю, что при создании системы нельзя снимать с себя обязанность обучать своих сотрудников, пытаясь вместо этого их подкупать. Суть управления заключается в том, чтобы создать такой порядок, при котором люди могут успешно решать задачи, при этом нужно избегать замены внутренней мотивации на внешнюю и злоупотребления карательными мерами и мелочным администрированием.

Расправившись с командно-административным и вульгарно-экономическим методами администрирования, рассмотрим еще один способ, к которому прибегают менеджеры, чтобы заставить людей двигаться в нужном им направлении. Я называю его методом управления «Отождествление». Подробнее я расскажу о нем в следующей главе.

Далее: Метод управления «Отождествление».

ГЛАВА СЕДЬМАЯ

Метод управления «Отождествление»

10 августа 2006 года, четверг

Как мы видели, при попытке организовать работу команды в едином направлении оказывается, что командно-административный и вульгарно-экономический методы управления весьма неэффективны, если это команда специалистов по высоким технологиям.

Но остается еще один метод, который я собираюсь назвать «методом управления «Отождествление» (Identity Management Method). Цель его в том, чтобы управлять, заставляя человека *отождествить* себя с задачей, которую он пытается решить. Этот метод похитрее предыдущих, и чтобы достичь с его помощью успеха, нужны незаурядные коммуникативные навыки. Но если действовать с умом, он работает лучше любого другого метода.

Проблема вульгарно-экономического метода в том, что он разрушает внутреннюю мотивацию. Метод управления «Отождествление» *создает* внутреннюю мотивацию.

Следуя методу управления «Отождествление», вы должны приложить все свои коммуникативные навыки, чтобы заставить своих работников отождествить свои задачи с задачами организации, что создаст для них высокую мотивацию, а затем вы снабдите их той информацией, которая им нужна для того, чтобы плыть в правильном направлении.

Как заставить людей отождествить себя с организацией?

Легче, когда задачи организации благородны или воспринимаются такими в силу каких-то обстоятельств. Apple создает почти фанатичное ото-

ждествление, почти целиком в рамках того сюжета, который начался с того рекламного ролика во время Суперкубка 1984 года: мы против тоталитаризма. Не очень отчетливая позиция, но действует. Мы в Fog Creek храбро выступили против умерщвления котят. Вот так!

Метод, который мне весьма нравится, – совместные обеды. Я всегда старался обедать вместе с коллегами. В Fog Creek предоставляются ежедневные обеды с обслуживанием для всей команды, во время которых мы сидим за одним общим столом. Трудно переоценить их значение для того, чтобы вся компания чувствовала себя одной семьей (в хорошем смысле). За шесть лет еще никто не отказался.

Возможно, мое признание позабавит некоторых летних практикантов, но одна из целей нашей программы производственной практики – заставить людей почувствовать себя жителями Нью-Йорка, чтобы им было легче привыкнуть к мысли о переезде сюда после колледжа для работы на полной ставке. Для этого у нас есть довольно плотная программа различных внеурочных мероприятий: два бродвейских шоу, «восхождение на скалу», речная прогулка вокруг Манхэттена, матч с участием Yankees, день открытых дверей, когда они могут встретить еще больше нью-йоркцев, посещение музея. Мы с Майклом устраиваем вечеринки у себя дома не только из гостеприимства, но и чтобы они увидели нью-йоркские квартиры, а не только общежитие, куда мы их поселили.

В целом, метод управления «Отождествление» требует создания единой крепкой команды, члены которой ощущают себя одной семьей, относясь к своим коллегам с чувством верности и долга.

Теперь вторая часть метода – дать людям информацию, нужную для движения организации в правильном направлении.

Сегодня Бретт зашел ко мне в кабинет, чтобы обсудить дату поставки FogBugz 6.0. Он склонялся к тому, чтобы назначить ее на апрель будущего года, меня же больше устраивал декабрь нынешнего. Конечно, до апреля мы могли более тщательно вылизать продукт и многое в нем улучшить; при поставке в декабре, скорее всего, пришлось бы отказаться от нескольких удачных новых функций.

Однако я объяснил Бретту, что весной нам потребуются шесть новых сотрудников, и шансов на то, что мы сможем их нанять, без FogBugz 6.0 значительно меньше. Таким образом, во время совещания с Бреттом я постарался объяснить ему, из каких финансовых соображений более ранняя поставка предпочтительнее, и уверен, что, ознакомившись с ними, он примет правильное решение... не обязательно совпадающее с *моим*. Если наши

продажи возрастут и без FogBugz 6.0, то, помня основные финансовые соображения, Бретт решит, что можно отложить выпуск версии 6.0, пока в нее не будут включены дополнительные функции. Смысл в том, что, поделившись с Бреттом информацией, я даю ему возможность принять правильное для Fog Creek решение, даже если изменятся обстоятельства. Если бы я попытался надавить на него, предложив денежную премию за каждый день сокращения сроков поставки по сравнению с апрелем, у него появилось бы желание продавать неотлаженную версию, имеющуюся *на данный момент*. Если бы я в командно-административном стиле потребовал, чтобы он, черт возьми, выпустил в установленный мной срок отлаженную версию, – возможно, он и сделал бы это, но возненавидел бы свою работу и уволился.

Заключение

Сколько менеджеров – столько и стилей управления. Я определил три основных метода: два простых и недейственных и один сложный, действенный, но на самом деле во многих программистских организациях управление осуществляется, скорее, специфическими, «показавшими действенность» методами, которые в разное время и у разных людей могут быть разными.

ЧАСТЬ ВТОРАЯ

Советы будущим программистам

ГЛАВА ВОСЬМАЯ

Опасности обучения на Java

29 декабря 2005 года, четверг

Ленивые детки.

Почему никто не хочет трудиться?

Верный признак начала старения: я начинаю ворчать и жаловаться, что «молодежь теперь не та» и что она не хочет или не может работать с полной отдачей.

Когда *я* был молодым, я учился программировать на перфокартах. Раз ошибешься – и карта испорчена, начинай все заново. У нас не было этих современных штучек, вроде клавиши Backspace, чтобы исправить ошибку.

С 1991 года на собеседованиях с программистами я обычно позволяю им выбрать для решения моих задачек любой язык программирования, который понравится. Тогда они в 99% случаев выбирали C.

Теперь они предпочитают писать на Java.

Поймите меня правильно: я вовсе не против использования Java в качестве языка реализации.

Пожалуй, стоит пояснить предыдущее высказывание. *В контексте настоящего обсуждения* я не собираюсь доказывать, что Java как язык реализации обладает недостатками. Эти недостатки многочисленны, но мы поговорим о них как-нибудь в другой раз.

Сейчас я имею в виду вот что: Java в целом недостаточно сложный язык программирования, чтобы позволить отличить замечательного програм-

миста от рядового. Этот язык может прекрасно подойти для каких-то работ, но речь сейчас не о том. Я бы даже сказал, что недостаточная сложность Java – это «фича, а не баг», тем не менее с этим языком связана отмеченная мной проблема.

Это дерзкое утверждение – результат моего наблюдения: две традиционные темы университетского курса программирования – указатели и рекурсия – многим учащимся оказываются не по зубам.

Раньше в колледже сначала читался курс по структурам данных – связанным спискам, хеш-таблицам и прочим типам, широко задействующим указатели. Такие курсы часто использовались для отсева: они были настолько сложны, что тот, кто с ними не справлялся, не мог рассчитывать на получение диплома по вычислительной науке, ибо если для вас слишком сложны указатели, то вряд ли вы одолеете теорию неподвижной точки!

Многие ребята из тех, кто в старших классах успешно писал для своего Apple II программы на BASIC, гоняющие по экрану мячик, записывались на курс «CompSci 101» (структуры данных), но когда дело доходило до указателей, их мозг перегревался, и вскоре они находили себе новую специальность вроде политологии, понимая, что юридический факультет – более удачный выбор. Я изучал статистику отсева среди студентов-кибернетиков – обычно это 40–70%. В университетах это считают потерями, но мне кажется, что отсеивать тех, кому программирование не доставит ни радости, ни успеха, просто необходимо.

Другим трудным начальным курсом для студентов-кибернетиков оказывался курс функционального программирования, в том числе рекурсивного. Очень высокий уровень этих курсов был установлен в MIT[1], где имелся обязательный курс (6.001) и учебник «Structure and Interpretation of Computer Programs»[2] (Abelson, Sussman) [The MIT Press, 1996]) – десятки, если не сотни лучших факультетов вычислительной техники использовали их как стандартное введение в специальность. (Вы можете – и должны – посмотреть старые варианты этих лекций в Сети.)

Сложность такого курса впечатляет. На первой же лекции вы узнаете достаточно много о Scheme и тут же изучаете функцию с неподвижной точкой, которая принимает на входе другую функцию. Посещая подобный курс CSE 121 в Пенне, я видел, что многим, если не большинству моих од-

---

[1] MIT – Massachusetts Institute of Technology (Массачусетский технологический институт). – *Прим. перев.*

[2] «Структура и интерпретация компьютерных программ».

нокурсников, он просто не по плечу. Я отправил преподавателю по электронной почте длинное письмо, пытаясь доказать несправедливость такого положения. Должно быть, кто-то в Пенне прислушался ко мне (или к другому жалобщику), потому что сейчас этот курс стали преподавать на Java.

Зря они это сделали.

Тут и зарыта собака. Многолетнее нытье ленивых студентов вроде меня и поступающие с предприятий жалобы на нехватку выпускников-программистов возымели действие, и в последние десять лет многие достойные в прочих отношениях учебные заведения почти повсеместно перешли на Java. Это модно, нравится кадровым агентам, которые оценивают резюме по методу «grep», но, главное, в Java нет сложностей, реально помогающих отсеивать программистов без той части мозга, которая распоряжается указателями и рекурсией, поэтому процент отсева снизился, на факультетах вычислительной техники стало больше студентов и увеличились бюджеты, так что довольны все.

Счастливчики, которых учат на Java, никогда не получат неожиданную ошибку сегментирования памяти, пытаясь реализовать хеш-таблицу, основанную на указателях. Им не придется ломать голову, пытаясь поразрядно упаковать данные. У них не встанут дыбом волосы от того, что в чисто функциональной программе, где значение переменной никогда не меняется, оно, тем не менее, постоянно меняется. Парадокс!

Они смогут получить высший балл по основной специальности и при отсутствии этой части мозга.

Может, я просто старый ворчун – из тех, кто гордится своим упорством, позволившим выбиться в люди несмотря на трудное детство?

Черт возьми, в начале прошлого века обязательными предметами в колледже были латынь и греческий – не потому, что от них было много пользы, а потому что считалось, что образованному человеку положено их знать. В некотором смысле мои аргументы аналогичны тем, которые приводились в пользу изучения латыни. «[Латынь] тренирует ум и память. Разгадывание латинских фраз – отличное упражнение для мышления, настоящий интеллектуальный пазл и хорошая тренировка логики», – пишет Скотт Баркер (Scott Barker, *www.promotelatin.org/whylatin.htm*). Но я не знаю ни одного университета, в котором сейчас была бы обязательна латынь. Может быть, указатели и рекурсия – это латынь и греческий вычислительной науки?

Я вполне готов признать, что 90% сегодняшнего кода обходится без указателей, а применять их в производственных программах просто опасно. Все правильно. И собственно функциональное программирование на практике используется не так уж часто. Согласен.

И все же владеть этими предметами необходимо в ряде самых интересных областей программирования. Например, не зная указателей, вы никогда не сможете работать с ядром Linux. Вы не поймете ни строчки кода Linux – да, фактически, любой операционной системы, – если не будете хорошо разбираться в указателях.

Тот, кто не понимает функционального программирования, не смог бы изобрести MapReduce – алгоритм, благодаря которому Google работает с гигантскими объемами данных, размещенных на многочисленных серверах. Термины «Map» и «Reduce» пришли из Lisp и функционального программирования. Понять, как работает MapReduce, может всякий, кто помнит из своего курса 6.001 или аналогичного ему, что у чисто функциональных программ нет побочных эффектов, поэтому их просто распараллеливать. Тот факт, что в Google смогли придумать MapReduce, а в Microsoft – нет, отчасти объясняет, почему Microsoft по-прежнему отстает в своих попытках наладить базовые функции поиска, тогда как Google уже решает новую задачу – построить Skynet^H^H^H^H^H^H, крупнейший в мире суперкомпьютер с с массовым параллелизмом. Я не уверен, что в Microsoft в достаточной мере понимают, как сильно они отстали в этом отношении.

А если копнуть чуть глубже, то действительная ценность указателей и рекурсии проявляется в том, что при создании крупных систем требуются гибкость ума, достигаемая в результате их изучения, и умственные способности, позволяющие избежать отсева после курсов, на которых они преподаются. Указатели и рекурсия требуют определенных способностей к рассуждению и абстрактному мышлению, а главное – умения рассматривать проблему на нескольких уровнях абстракции одновременно. Поэтому способность разобраться в указателях и рекурсии непосредственно связана с перспективой стать выдающимся программистом.

Преподавание вычислительных наук только на примере Java не позволяет отсеять студентов, у которых не хватает живости ума, необходимой для работы с данными понятиями. Как работодатель я вижу, что многие обладатели дипломов CS – выпускники заведений с полным обучением на Java – как программисты просто ни на что не способны, кроме как написать очередную бухгалтерскую программу на Java, хотя им и удалось проскочить через эту новую оглупленную систему курсов. Они ни за что не

справились бы с 6.001 в MIT или CS 323 в Йеле и, честно говоря, это одна из причин, по которым для меня как работодателя диплом CS из MIT или Йеля «весит» больше, чем диплом CS из Дьюка, где недавно полностью перешли на Java, или из Пенна, где Scheme и ML заменили на Java в том курсе, который когда-то чуть не доконал меня с друзьями, – CSE 121. Это не значит, что я не возьму к себе толковых ребят из Дьюка или Пенна, – возьму, но просто мне будет гораздо труднее оценить их реальные способности. Раньше я определял способных ребят по тому, как они за считанные секунды разбирались в рекурсивном алгоритме или, не задумываясь, писали функцию обработки связанных списков на основе указателей. Но столкнувшись с выпускником Java-колледжа, я не могу понять, чем вызваны его затруднения, – недостатком образования или отсутствием того участка мозга, который необходим для решения сложных задач программирования. Пол Грэм (Paul Graham, *www.paulgraham.com/avg.html*) называет их «дутыми программистами» (blub programmers).

Жаль, что Java-колледжи не отсеивают тех, из кого никогда не получится отличный программист, – хотя подобные заведения и оправдываются тем, что это не их проблема. Производство или, во всяком случае, кадровики, работающие методом «grep», в открытую требуют, чтобы студентов обучали Java.

Кроме того, Java-колледжи не развивают мозг своих студентов, и в результате им не достает ловкости, живости и гибкости ума, требующихся для того, чтобы хорошо проектировать программное обеспечение (я не имею в виду объектно-ориентированное «проектирование», в котором уйма времени уходит на бесконечное перетряхивание иерархии объектов или мучения по поводу надуманных проблем типа «как правильно – has-a или is-a?»). Нужно учиться мыслить одновременно на нескольких уровнях абстракции, и это именно тот тип мышления, который требуется для проектирования выдающихся программных архитектур.

Может ли изучение объектно-ориентированного программирования (ООП) стать адекватной заменой указателям и рекурсии в смысле отсева? Если ответить коротко, то нет. Не касаясь достоинств ООП, просто скажу – оно не настолько сложно, чтобы отсеивать посредственных программистов. Обучение ООП заключается, в основном, в заучивании ряда терминов вроде «инкапсуляции» или «наследования» и тестов с вариантами ответов, где нужно правильно выбрать между полиморфизмом и перегрузкой. Не труднее, чем запомнить несколько дат и имен в курсе истории. Проблема в ООП – это когда *ваша программа работает*, но ее трудно сопровож-

дать. Якобы. А проблема с указателями – это когда ваша программа выдает ошибку «Segmentation Fault», и у вас нет ни малейшего понятия о том, что случилось, пока вы не сделаете глубокий вдох и не заставите свой мозг работать на двух уровнях абстракции одновременно.

Я неспроста высмеиваю здесь кадровиков, действующих по методу «grep». Я еще не встречал такого программиста, который, зная Scheme, Haskell и указатели C, не смог бы за пару дней разобраться с Java и начать писать на нем программы лучше обладателей пятилетнего опыта работы с Java, – но попробуйте объяснить это типичному зануде из отдела кадров!

Какие же цели ставят перед собой факультеты вычислительных наук? Это же не профессионально-технические училища! Не их дело готовить рядовых работников для отрасли – для этого есть местные колледжи и государственные программы переподготовки для безработных. Университет должен давать студенту фундаментальные знания на всю жизнь, а не готовить его к тому, чтобы найти какую-то работу. Разве не так?

Итак. Вычислительная наука – это доказательства (рекурсия), алгоритмы (рекурсия), языки (лямбда-исчисление), операционные системы (указатели), компиляторы (лямбда-исчисление), откуда следует, что в Java-колледже, где не преподают C и Scheme, фактически не преподают вычислительную науку вообще. В реальном мире могут считать обсасывание понятия функции бесполезным занятием, но очевидно, что это совершенно необходимо для аспирантуры по вычислительной науке. Мне непонятно, как профессора вычислительной науки, составляя учебные планы, позволили программам своих учебных заведений опуститься до такого низкого уровня, когда выпускаются программисты только для производства, а не те, которые смогли бы окончить аспирантуру, получить докторскую степень и когда-нибудь занять их место. Хотя, погодите. Кажется, я понял.

На самом деле, если вспомнить дискуссии, которые велись в академических кругах во времена «Великого перехода на Java», то можно заметить: самым большим опасением было то, что Java – недостаточно простой язык для использования в процессе обучения.

Боже милостивый, подумалось мне, *да ведь они хотят еще больше упростить учебные планы!* Кормить студентов с ложечки! Пусть вдобавок ассистенты сдают зачеты вместо студентов, и тогда обучение в Америке перестанет кого-либо интересовать. Как можно чему-нибудь научиться, если учебный план тщательно составлен с целью до предела все облегчить? Кажется, есть даже специальная группа, которая должна выделить из Java достаточно простое подмножество, чтобы преподавать его учащимся,

и создать упрощенную документацию, скрывающую от неокрепших умов весь этот хлам EJB/J2EE, чтобы они не забивали свои головки разными курсами, которые им не понадобятся для решения все более простых вычислительных задач.

Самый мягкий ответ на вопрос, почему факультеты вычислительных наук так стремятся упростить свои курсы, заключается в предположении, что они надеются выкроить дополнительное время на теоретические понятия, если не придется целых две лекции объяснять студентам разницу между, скажем, Java int и Integer. Если это действительно так, то курс 6.001 – идеальное решение: Scheme – настолько простой учебный язык, что толковые студенты могут целиком освоить его за 10 минут, а оставшуюся часть семестра можно посвятить неподвижным точкам.

Фу.

Возвращаюсь к нулям и единицам.

(Вы изучали единицы? Повезло! У нас были только нули.)

ГЛАВА ДЕВЯТАЯ

Лекция в Йеле

3 декабря 2007 года, понедельник

Часть лекции, прочитанной на факультете вычислительных наук в Йеле 28 ноября 2007 года.

Я получил степень бакалавра вычислительной науки (computer science, CS) в 1991 году. Шестнадцать лет назад. Сегодня я попытаюсь связать старшие курсы моего обучения на факультете вычислительной науки со своей профессиональной деятельностью, в которую входят разработка программного обеспечения, публикации о программном обеспечении и основание софтверной компании. Конечно, все это немного нелепо: изданное в MIT «Introduction to Computer Science» («Введение в вычислительную науку») Хэла Абельсона (Hal Abelson) начинается с известного авторского разъяснения, что «вычислительная наука» – это не наука о компьютерах, и что это вообще не наука, поэтому с моей стороны было бы дерзостью утверждать, что CS – лучшая подготовка к карьере разработчика программного обеспечения, чем, скажем, изучение средств массовой информации или культурной антропологии.

Тем не менее я продолжу свою мысль. Одним из наиболее полезных курсов оказался тот, который я бросил после первой лекции. Другим был курс Роджера Шанка (Roger Schank), настолько презираемый на факультете, что его не учитывали при получении степени по вычислительной науке. Но об этом чуть позже.

Третьим был курс под названием CS 322, который теперь называется CS 323. В мои времена CS 322 требовал столько сил, что на него давали

1,5 кредит-часа. А в Йеле лишние 0,5 кредит-часа учитываются только с другими 0,5 кредит-часа на том же факультете. Действительно, было еще два курса по 1,5 кредит-часа, но можно было записаться только на оба сразу. Благодаря такому мошенничеству 0,5 кредит-часа просто не учитывались, но этим оправдывались еженедельные наборы задач, на решение которых уходило 40 часов. После многолетних жалоб студентов курс был переоценен в 1 кредит-час, получил номер CS 323 и по-прежнему еженедельно требовал 40 часов на решение задач. В остальном он остался примерно тем же. Я его любил, потому что я люблю программировать. Лучше всего в CS 323 то, что благодаря этому курсу до многих доходит, что они никогда не станут программистами. Это его достоинство. Те, кому не выпало узнать на уроках Стэна Эйзенштата (Stan Eisenstat), что программистами им не стать, влачат жалкую карьеру, вырезая и вставляя куски Java-кода. Кстати, для тех, кто записался на CS 323 и получил на экзамене оценку «A»: у нас в Fog Creek есть замечательная летняя практика. Подойдите ко мне после лекции.

Насколько я могу судить, базовый список курсов остался прежним. 201, 223, 240, 323, 365, 421, 422, 424, 429 – это почти те же курсы, которые читались шестнадцать лет назад. С тех пор, как я покинул Йель, количество выбравших специальность CS на самом деле выросло, хотя из-за временного всплеска времен дотком-бума кажется, что оно снизилось. И стало гораздо больше интересных факультативных курсов, чем в мои времена. Стало быть, прогресс налицо.

Кстати, я ведь тогда собирался получить степень PhD. Оба моих родителя – преподаватели. Среди их друзей множество ученых, и в детстве мне казалось, что все взрослые непременно получают PhD. Во всяком случае, я всерьез подумывал о поступлении в аспирантуру по вычислительной науке. Пока не попытался прослушать курс динамической логики – здесь, на этом самом факультете. Его читала Ленора Зак (Lenore Zuck), которая сейчас работает в UIC.

Меня хватило ненадолго, а из услышанного я понял немного. Насколько я припоминаю, динамическая логика похожа на формальную: Сократ – человек, все люди смертны, следовательно, Сократ тоже смертен. Разница в том, что в динамической логике истинные высказывания со временем могут меняться. Сократ был человеком, теперь он кот и т. д. Теоретически, это должен быть интересный способ доказательства утверждений о компьютерных программах, где состояния, т. е. истинные значения, меняются во времени.

На первой лекции Ленора Зак предложила несколько аксиом и правил преобразования и начала доказывать довольно простую вещь. Она взяла компьютерную программу `f := not f`, где `f` – булево значение, бит которого просто менялся на противоположный; требовалось доказать, что если выполнить эту программу четное количество раз, то в результате `f` будет иметь то же значение, что и вначале.

Доказательство длилось довольно долго. Если мне не изменяет память, все происходило как раз в этом зале (ковер здесь все тот же), и вот эти доски были исписаны доказательствами. Профессор Зак применяла доказательство методом индукции, методом от противного, методом перебора всех вариантов – курс читался в конце дня, и мы уже минут на сорок задержались – и методом привлечения старшекурсника, когда она говорит «не могу вспомнить, как доказать вот этот переход», а старшекурсник из передних рядов отвечает: «Да-да, все правильно, профессор».

И когда все уже было сделано и записано, в конце доказательства она умудрилась получить результат, прямо противоположный здравому смыслу, и тогда тот же старшекурсник указал место, в котором 63 шага назад один бит случайно инвертировался из-за грязи на доске, и все стало на свои места.

В качестве домашнего задания она предложила доказать обратное: если выполнить программу `f := not f` n раз, и бит окажется в начальном состоянии, то `n` должно быть четным.

Я решал эту задачу несколько часов. Передо мной было ее доказательство в прямом направлении, где при ближайшем рассмотрении обнаружилось множество отсутствующих переходов – «очевидных», но не для меня. Я прочел все, что можно было найти о динамической логике в Becton Center, и сражался с задачей до глубокой ночи. Я нисколько не продвинулся и все больше разочаровывался в теоретической вычислительной науке. Я пришел к выводу, что если доказательство занимает многие страницы, то вероятность ошибок в нем значительно выше, чем вероятность ошибки собственной интуиции относительно тривиального утверждения, которое нужно доказать. Я решил, что все эти штучки с динамической логикой – не самый плодотворный способ доказывать факты, касающиеся настоящих, интересных компьютерных программ, потому что гораздо легче ошибиться в доказательстве, чем в интуитивном представлении о том, как работает программа `f := not f`. Поэтому я бросил этот курс – слава богу, что было время присмотреться, – более того, я тут же решил, что аспирантура

по вычислительной науке не для меня, так что этот курс оказался самым полезным из всех, на какие я когда-либо записывался.

Это привело меня к одному важному рабочему наблюдению. Бывает, что программист переопределяет задачу, так что она становится доступной для алгоритмического решения. Переопределив задачу, он часто получает нечто доступное для решения, но, по сути, тривиальное. Он не решает реальную задачу, потому что та неразрешима. Приведу пример.

Часто говорят о своего рода кризисе качества разработки программного обеспечения. Я не вполне согласен с такими заявлениями: компьютерные программы, используемые большинством людей, обладают на редкость высоким качеством в сравнении со всем прочим, что их окружает, но речь сейчас не о том. Тезис о «кризисе качества» порождает массу предложений и исследований по улучшению качества программного обеспечения. С этой точки зрения людей можно разделить на две категории – «гики» и «пиджаки» (geeks and suits).

Гики хотят решать проблему автоматически, с помощью программ. Они предлагают различные способы «доказательства», что в программе нет ошибок, – модульное тестирование, управляемая тестированием разработка, автоматическое тестирование, динамическая логика и так далее.

Пиджаки не вполне понимают, в чем проблема. Их совершенно не волнует, что в программе есть ошибки, коль скоро люди ее покупают.

В данный момент в борьбе между гиками и пиджаками побеждают последние, поскольку они управляют бюджетами и, честно говоря, я не вижу в этом ничего ужасного. Пиджаки знают, что устранение ошибок подчиняется закону уменьшающейся прибыли. Если программе обеспечен определенный уровень качества, благодаря чему некоторые пользователи смогут решать свои задачи, значит, они купят эту программу и будут получать с ее помощью прибыль.

Кроме того, пиджаки шире рассматривают понятие «качество». У них абсолютно корыстный подход: качество программы определяется тем, насколько она может повысить их премию по результатам года. Кстати, тут учитывается не только отсутствие ошибок в программе. Например, очень ценится включение в программу новых функций для решения новых задач новыми пользователями, что позволяет гикам презрительно называть программы «распухшими от функций» (bloatware). Учитывается и эстетическая сторона: красивая программа продается лучше, чем безобразная. Учитывается степень удовлетворенности пользователя. В принципе, такой

подход дает пользователям возможность самим определять, в чем состоит качество, и удовлетворяет ли программа их потребностям.

А гиков интересуют узкотехнические аспекты качества. Они смотрят в код, а не ждут, что скажут пользователи. Будучи программистами, они стремятся автоматизировать все вокруг, включая, конечно, процедуру контроля качества. Отсюда поблочное тестирование (unit testing), что не так уж плохо, и отсюда все попытки механически «доказать», что программа «корректна». Беда в том, что из определения качества выбрасывается все не поддающееся автоматизации. Мы знаем, что пользователи предпочитают программы со стильным интерфейсом, но поскольку автоматически измерить стильность нельзя, это свойство остается за рамками автоматизированной процедуры контроля качества.

На практике мы видим, что самые консервативные гики стараются отказаться от всех полезных оценок качества и оставить лишь то, что они могут проверить механически, то есть поведение программы согласно спецификации. В итоге мы получаем узкотехническое определение качества: степень соответствия программы ее спецификации. Выдает ли она правильный результат, получив на входе правильные данные?

Это очень существенная проблема. Чтобы механически проверить соответствие программы некоторой спецификации, эта спецификация должна быть предельно подробной. Фактически, такая спецификация должна полностью определять всю программу, иначе ничего нельзя будет доказать автоматически и механически. Теперь смотрите: если спецификация полностью определяет то, как должна вести себя программа, то в ней достаточно информации для того, чтобы сгенерировать саму программу! И тогда некоторые гики заходят настолько далеко, что подумывают об автоматической компиляции спецификаций в программы, и начинают думать, что изобрели способ программировать, не прибегая к программированию.

Но ведь это эквивалент вечного двигателя для разработки программного обеспечения. Всегда найдется очередной псих-изобретатель, сколько ни объясняй, что это невозможно. Если спецификация точно определяет, что будет делать программа, и настолько подробно, что из нее можно сгенерировать саму программу, напрашивается вопрос: а где взять такую спецификацию? Написать полную спецификацию столь же сложно, как соответствующую ей программу, потому что составитель спецификации должен так же детально решить вопросы, как и программист. В терминах теории информации: спецификация должна обладать точно таким же показателем шенноновской энтропии, как и сама программа. Каждый бит

энтропии – это решение, принятое составителем спецификации или программистом.

Итог таков: если найдется механический способ проверки корректности программы, то все, что вы сможете доказать, – это идентичность этой программы некой другой программе с таким же показателем энтропии, как у первой; в противном случае какие-то виды поведения окажутся не определенными, а значит, не будут проверены. Поскольку написать спецификацию так же трудно, как саму программу, все сводится к перемещению проблемы из одного места в другое без какого-либо фактического результата.

Может, это и грубый пример, тем не менее поиск Святого Грааля качества программ многих заводит в разного рода тупики. В подобном положении в Microsoft оказалась команда разработчиков Windows Vista. По-видимому, – судя по слухам и намекам в блогах – у Microsoft есть долгосрочная политика искоренения тестеров, не умеющих писать код, и замены их так называемыми SDET (Software Development Engineers in Test) – программистами, пишущими скрипты для автоматизированного тестирования.

Прежние тестеры Microsoft проверяли множество вещей: единообразие и читаемость шрифтов, разумное и аккуратное размещение элементов управления в диалоговых окнах, мерцание экрана во время работы, интерфейс пользователя, легкость работы с программой, единый стиль сообщений. Их интересовала скорость работы программы, отсутствие орфографических и грамматических ошибок в сообщениях, и массу времени они тратили на проверку единообразия интерфейса в разных частях программы, потому что когда интерфейс выдержан в едином стиле, им легче пользоваться.

Все это нельзя проверить автоматическими скриптами. Поэтому одним из результатов новой политики автоматизированного тестирования было то, что Windows Vista оказалась крайне непоследовательной и полной шероховатостей. В окончательном продукте есть масса очевидных проблем... ни одна из которых не является «ошибкой» с точки зрения автоматизированных сценариев, но в совокупности они создают впечатление, что Vista – это шаг назад в сравнении с XP. Определение качества, данное гиками, возобладало над определением пиджаков. Я уверен, что автоматизированные сценарии тестирования для Windows Vista успешно выполняются в Microsoft, но что в этом толку, если чуть ли не каждый технический писатель рекомендует пользователям как можно дольше держаться за XP. По-

видимому, никто не написал автоматизированный тест для проверки наличия у пользователей XP достаточных оснований перейти на Vista.

Я отнюдь не испытываю ненависти к Microsoft. Это мое первое рабочее место после университета. В те времена это было недостаточно респектабельно, как устроиться в цирк. На тебя смотрели с удивлением. Microsoft? В самом деле? У нас в кампусе, в частности, считалось, что это такая скучная корпорация, где нужно ходить застегнутым на все пуговицы и писать скучное программное обеспечение, позволяющее бухгалтерам создавать таблицы – или чем там они занимаются? Совершенно не впечатляет. И все это для жалкой однозадачной операционной системы MS-DOS, в которой полно непонятных глупых ограничений вроде восьмисимвольных имен файлов, нет электронной почты, нет telnet, нет телеконференций Usenet. Давно уже нет MS-DOS, но культурная пропасть между юниксоидами и пользователями Windows все глубже. Разногласия очень запутанные и очень фундаментальные. В Йеле Microsoft считалась местом, где пишут операционные системы для производства игрушек с применением вычислительной науки тридцатилетней давности. А в Microsoft поднимали на смех новичков со странными гипотезами вроде той, что следующим основным языком программирования станет Haskell, обзывая их «компьютерными шаманами».

Приведу лишь один маленький пример культурной несовместимости UNIX и Windows. Элементом культуры UNIX является отделение пользовательского интерфейса от функциональности. Уоригинальной системы UNIX изначально есть только интерфейс командной строки, но если повезет, кто-нибудь напишет для нее милый интерфейс со всеми эффектами теней, полупрозрачности и трехмерности, и эта прелестная клиентская часть незаметно для пользователя запустит интерфейс командной строки, а он таинственным образом даст сбой, не отражающийся правильно в красивом пользовательском интерфейсе, и тот зависнет в ожидании ввода пользователем данных, которых так никогда и не получит.

Утешает то, что интерфейс командной строки можно использовать в скриптах.

Культура Windows предполагает, что сначала пишется приложение с GUI[1], а потом все важнейшие функции безнадежно перемешиваются с кодом интерфейса пользователя и получаются гигантские приложения

[1] *GUI* (Graphical User Interface) – графический интерфейс пользователя. – *Прим. перев.*

вроде Photoshop, позволяющего великолепно отредактировать фотографию, но если вы программист, и вам нужно с помощью Photoshop изменить размер 1000 фотографий, так чтобы их высота не превышала 200 пикселов, вы не сможете написать нужный код, потому что все очень тесно связано с интерфейсом пользователя.

Во всяком случае, эти две культуры примерно соответствуют противостоянию между интеллектуалами и обывателями, что находит точное отражение в учебных планах различных факультетов вычислительной науки. В институтах Ivy League вы найдете только UNIX, функциональное программирование и теорию конечных автоматов. С понижением уровня запросов все чаще встречается Java. Если опуститься еще ниже, точно появятся курсы с темами вроде «Microsoft Visual Studio 2005 101», три кредит-часа. Ну, а добравшись до двухгодичных заведений, вы увидите те самые курсы «сертификации» типа «SQL Server за 21 день», какие рекламируются по выходным дням на кабельном ТВ. Ваша карьера начнется (другим голосом) с Java Enterprise Beans!

4 декабря 2007 года, вторник

Это вторая часть лекции, прочитанной 28 ноября 2007 года на факультете вычислительных наук в Йеле.

Проведя несколько лет в Редмонде (штат Вашингтон) в бесплодных попытках приспособиться к окружающей среде, я сбежал в Нью-Йорк. Там я несколько месяцев продолжал работать на Microsoft, в качестве совершенно бездарного консультанта Microsoft Consulting, после чего несколько лет в 1990-х, в эпоху зарождения Интернета, проработал в Viacom. Это крупный конгломерат, владевший MTV, VH1, Nickelodeon, Blockbuster, Paramount Studios, Comedy Central, CBS и целой кучей других компаний индустрии развлечений. В Нью-Йорке я впервые разобрался в том, как зарабатывает себе на жизнь большинство программистов. Это жуткая вещь под названием «внутрифирменное ПО» (in-house software). Ужас, которого нужно постараться избежать. Вы работаете программистом в большой компании, которая производит, скажем, алюминиевые банки, но никакой готовый продукт не обрабатывает алюминиевые банки в точности так, как надо, поэтому у компании есть собственные программисты или она нани-

мает по завышенным ценам программистов у таких компаний, как Accenture и IBM, чтобы те написали им нужное программное обеспечение. Это ужасно по двум причинам: во-первых, это не приносит вам удовлетворения как программисту по ряду оснований, которые я приведу ниже, а, во-вторых, это ужасно потому, что 80% связанных с программированием работ именно таковы, и если в конце обучения не проявить крайнюю степень осторожности, вы как раз окажетесь разработчиком внутрифирменного ПО, и, смею вас заверить, такая работа лишит вас всякого интереса к жизни.

Так что же ужасного в том, чтобы разрабатывать собственное программное обеспечение для фирмы?

Первое: вам никогда не дадут делать что-либо правильно. Вы всегда будете вынуждены действовать практически выгодным образом. Нанять такого программиста недешево: обычно компании типа Accenture или IBM берут 300 долларов в час за услуги недавнего выпускника Йеля с дипломом политолога, который прослушал шестинедельный курс .NET-программирования, зарабатывает 47 000 долларов в год и надеется на хорошую практику, чтобы попасть в школу бизнеса, – иными словами, за большие деньги компания получает программиста, которому не разрешат работать с Ruby on Rails, как бы крут ни был Ruby и каким бы классным ни стал Ajax. Вы откроете Visual Studio, запустите мастер, перетащите на страницу элемент управления Grid, подключите его к базе данных – и готово. Достаточно. Переходим к следующей задаче. Вот и вторая причина, по которой эта работа так противна: как только программа начинает достаточно прилично работать, ее разработка заканчивается. Основные функции действуют, главная задача решена, и дальше никакого возврата на инвестиции, никаких деловых причин, чтобы пытаться ее улучшить. Поэтому все доморощенные программы выглядят тошнотворно: никто и копейки не потратит на то, чтобы они выглядели лучше. Можете забыть про гордость профессией или ремеслом, воспитанную в вас курсом CS 323. Вы будете со стыдом плодить хлам, а потом вас бросят на починку прошлогоднего хлама, который стал разваливаться – главным образом, потому, что сделан неправильно, – и через двадцать семь лет такой жизни вы получите золотые часы. Впрочем, золотые часы больше не раздают. Двадцать семь лет – и вы получите синдром запястного канала. К примеру, в компании, создающей товарный продукт или даже сетевой продукт типа Google или Facebook, чем лучше у разработчика получится этот продукт, тем лучше он будет продаваться. Главная же особенность внутрифирменного продукта в том, что как

только он стал «достаточно хорош», работа над ним прекращается. Товарный продукт можно бесконечно уточнять, шлифовать, перерабатывать и совершенствовать, например, работая для Facebook, вы можете месяц оптимизировать код, выбирающий имя через Ajax, пока он не станет действительно красивым и быстрым, и все эти труды оправданны, потому что делают продукт лучше, чем у конкурента. Итак, вторая причина, по которой лучше работать над товарным продуктом, чем над внутрифирменным, – это возможность делать красивые вещи.

Третья причина: когда вы работаете программистом в софтверной компании, ваша работа непосредственно связана с тем, каким образом ваша компания зарабатывает деньги. Из этого следует, например, что руководство компании заботится о вас. Вы получаете все виды вознаграждения, лучшие офисы и возможность продвижения по службе. Программист никогда не сможет стать руководителем Viacom, но в технической компании вы вполне можете дорасти до главы компании.

И тем не менее. После Microsoft я поступил на работу в Viacom, потому что хотел больше узнать об Интернете, а Microsoft в те времена его упорно игнорировала. Но в Viacom я был просто внутрифирменным программистом, на несколько уровней ниже любого, кто занимался тем, на чем Viacom зарабатывала деньги.

И, должен сказать, несмотря на всю важность правильного входа в Интернет для Viacom, когда дошло до расселения людей по помещениям, собственных программистов затолкали по трое в кабинку в дальнем конце, где окон и в помине не было, зато «продюсеры» (не знаю, чем занимались эти типы, напоминающие Черепаху из сериала «Антураж») получили собственные кабинеты с огромными окнами на Гудзон. Однажды на новогодней вечеринке Viacom я был представлен начальнику, отвечавшему за интерактивную стратегию или нечто в этом роде (очень высокий пост). Он сказал что-то туманное и глупое о важности интерактивности, за которой будущее, из чего я сделал вывод, что у него нет ни малейшего представления о происходящем, об Интернете или о том, чем я занимаюсь как программист, и его все это немного пугало, но не все ли равно, потому что он зарабатывает два миллиона в год, а я всего лишь печатаю на машинке, или какой-то «оператор HTML», или бог весть чем занимаюсь, с чем вполне справилась бы его дочь-школьница.

Итак, я перебрался в соседнее место под названием Juno Online Services. Это был один из первых интернет-провайдеров, предоставлявший бесплатный доступ по телефонным линиям для работы с электронной поч-

той – и ничего больше. Это отличалось от Hotmail и Gmail, которых тогда еще не было, потому что доступ к Интернету был не нужен, и система была действительно бесплатной.

Источником средств для Juno якобы была реклама. Оказалось, однако, что реклама, нацеленная на людей, не желавших платить AOL 20 долларов в месяц, – не самый доходный бизнес, так что в действительности Juno поддерживали богатые инвесторы. Но, во всяком случае, компания Juno производила продукт, программисты там были в почете, и меня удовлетворяла поставленная задача обеспечить всех электронной почтой. И я действительно с удовольствием проработал там три года программистом C++. Однако со временем я стал подумывать, что философия руководства Juno устарела. Считалось, что менеджеры нужны для того, чтобы объяснять людям, что они должны делать. Этот подход диаметрально противоположен принятому в компаниях высоких технологий на западном побережье. На Западе я привык считать, что руководить – значит выполнять неприятные и скучные обязанности, которыми кто-то должен заниматься, чтобы толковые люди могли делать свое дело. Сравните это с научным факультетом в университете, руководить которым – тяжкий труд, за который никто не хочет браться, предпочитая научные исследования. Таков стиль управления в Силиконовой долине. Менеджеры нужны для того, чтобы мебель не мешалась на дороге и подлинные таланты могли ваять свои шедевры.

Juno была основана очень молодыми и очень неопытными людьми: президенту компании было 24 года, и это была его первая работа, а не просто первая административная работа, – и из каких-то книг, фильмов или ТВ-шоу он почерпнул представление о том, что менеджеры нужны для того, чтобы ПРИНИМАТЬ РЕШЕНИЯ.

Единственное, в чем я уверен, так это в том, что менеджер меньше всех осведомлен в любом техническом вопросе и что это последний, кому можно доверить принимать решения. Когда я работал в Microsoft, сотрудники отделения, занимавшегося разработкой приложений, приходили к своему руководителю Майку Мэйплсу, чтобы он разрешил какой-нибудь возникший у них технический спор. Он перебрасывался с ними несколькими остротами, рассказывал анекдот, а потом посылал к черту, чтобы они сами решали свои вопросы, а не ходили к нему, потому что он не готов принимать решения, связанные с технической стороной дела. И я полагал, что только так и можно управлять умными и высококвалифицированными людьми. Но менеджеры Juno – как Джордж Буш – считали, что должны принимать решения, а решений нужно было принимать очень много, по-

этому они практиковали стиль, который я назвал микроуправлением путем набегов: они вдруг сваливались с неба, решали какой-то крохотный вопрос типа метода ввода даты в диалоговом окне, отвергая при этом мнения квалифицированных технических специалистов, занимавшихся данной проблемой в течение нескольких недель, а потом исчезали – почему я и говорю о набегах, – поскольку где-то еще возникла ничтожная проблема, требующая их микроуправления.

Итак, я ушел оттуда, не имея реального плана.

5 ДЕКАБРЯ 2007 ГОДА, СРЕДА

Это третья часть лекции, прочитанной 28 ноября 2007 года на факультете вычислительных наук в Йеле.

Я отчаялся найти работу в такой компании, где к программистам относились бы как к талантам, а не как к машинисткам, и решил, что нужно создать собственную фирму. В те времена мне попадалось множество очень глупых людей с очень глупыми бизнес-планами, которые создавали интернет-компании, и я решил, что если я хотя бы на 10% умнее их, что не исключено, то тоже смогу создать компанию, и уж в этой компании будут совсем другие порядки. Мы станем относиться к программистам с уважением, мы будем выпускать продукты высокого качества, мы не станем связываться с венчурным капиталом или 24-летними президентами, мы будем проявлять заботу о своих клиентах и решать их проблемы, когда они обратятся к нам за помощью, а не сваливать всю вину на Microsoft, и пусть наши клиенты сами решают, платить нам или не платить. В Fog Creek мы будем возвращать деньги всем желающим, не задавая никаких вопросов и при любых обстоятельствах. Мы будем вести праведную жизнь.

Это было летом 2000 года, и я на какое-то время отошел от работы, вынашивая планы создания Fog Creek Software и много времени проводя на пляже. Тогда-то я и стал записывать некоторые размышления по поводу своей карьеры и публиковать их на сайте «Joel on Software». Еще не были изобретены блоги, и некий программист по имени Дэйв Уайнер (Dave Winer) организовал систему под названием EditThisPage.com, позволявшую всем публиковаться в Сети в формате, напоминающем блог. «Joel on Software» быстро развивался и дал мне ту трибуну, с которой я мог излагать

свои мысли о разработке программного обеспечения, привлекая к ним внимание. Сайт состоял из малооригинальных идей, сдобренных шутками. Он имел успех, потому что я применял шрифт крупнее обычного, что облегчало чтение. Оценить, сколько людей читает сайт, достаточно трудно, особенно если не ставить себе целью их учет, но типичные статьи на этом сайте имели от 10 000 до миллиона читателей в зависимости от популярности той или иной темы.

Тому, чем я занимаюсь на «Joel on Software» – писанию статей на технические темы, – я тоже научился здесь, на факультете CS. Вот как это произошло. В 1989 году в Йеле достаточно успешно занимались искусственным интеллектом (ИИ), и один из светил в этой области, профессор Роджер Шанк (Roger Schank), приехал сюда и прочел небольшую лекцию о некоторых своих изысканиях в области ИИ – скрипты, схемы, слоты и все такое прочее. Сейчас я подозреваю, что эту самую лекцию он читал уже двадцатый год, и в течение двадцати лет своей карьеры он писал программки, основанные на его теориях, – вероятно, для их проверки, – и хотя они не работали, его теории почему-то никогда не отвергались. Он производил впечатление блестящего ученого, и я хотел записаться к нему на курс, но узнал, что он не любит студентов младших курсов, и единственная возможность – записаться на его курс под названием «Algorithmic Thinking» («Алгоритмическое мышление»), CS 115, – по существу, облегченный конспект курса группы IV, рассчитанный на поэтов. Формально он числился за факультетом CS, но отношение к нему в преподавательской среде было таково, что в зачет по специальности CS он не шел. Хотя это был самый посещаемый курс на факультете CS, меня коробило всякий раз, когда мои друзья, учившиеся на историков, называли его «вычислительной наукой». Типичным заданием было написать сочинение на тему «Могут ли думать машины». Теперь вы поняли, почему этот курс не засчитывался для получения степени по вычислительной науке. Не удивлюсь, если, узнав про этот курс, вы потребуете, чтобы у меня отобрали диплом.

Лучшим в курсе «Алгоритмического мышления» было то, что приходилось очень много писать. Было тринадцать контрольных работ – раз в неделю. Оценки не ставились. Нет, конечно, ставились. Впрочем, тут особая история. Одна из причин, по которым Шанк так не любил студентов младших курсов, была их озабоченность оценками. Ему хотелось поговорить о том, могут ли компьютеры думать, а они требовали объяснить, почему их работа получила оценку «B», а не «A». В начале семестра он произнес длинную речь о вредности оценок и объявил, что единственной оценкой на ва-

шей работе может быть маленькая галочка, означающая, что какой-то старшекурсник ее прочел. Со временем он решил отличать более удачные работы, на которых ставилась галочка с плюсом, а поскольку встречались и очень слабые работы, он ставил на них галочку с минусом. Припоминаю, что однажды я получил галочку с двумя плюсами. Но отметку – ни разу.

Несмотря на то что курс CS 115 не шел в диплом по специальности, самым полезным, что я вынес с факультета CS, оказался полученный опыт написания статей на темы, связанные с техникой. Умение ясно писать на технические темы – это то, что отличает руководителя от невнятно бормочущего программиста. Первая моя должность в Microsoft – руководитель программы в команде Excel, и, занимая ее, я написал техническую спецификацию для огромной программной системы под названием Visual Basic for Applications. В этом документе было около 500 страниц, и каждый день буквально сотни людей, придя на работу, читали в нем, что они должны сегодня делать. Среди этих людей были программисты, тестеры, маркетологи, составители документации и специалисты по локализации со всех концов света. Я заметил, что выдающимися менеджерами программ в Microsoft становились те, кто умел действительно хорошо писать. Microsoft повернула свой стратегический курс на 180 градусов в результате одного лишь неотразимого электронного письма, которое написал Стив Синофски (Steve Sinofsky), озаглавив его «Cornell is Wired» («Весь Корнелл в сети!», *www.cornell.edu/about/wired/*). Исход спора решает тот, кто умеет писать. Язык программирования C обрел свое влияние благодаря замечательной книге Брайана Кернигана (Brian Kernighan) и Денниса Ричи (Dennis Ritchie) «The C Programming Language» (Prentice Hall, 1988).

Итак, вот кульминационные моменты моего обучения вычислительным наукам: CS 115, где я научился писать, единственная лекция по динамической логике, благодаря которой я не пошел в аспирантуру, и CS 322, где я изучил обряды и ритуалы учения UNIX и с удовольствием написал массу кода. Главное, чему не учат, готовя к степени по компьютерной науке, – это тому, как разрабатывать программы, хотя у вас и развиваются определенные части мозга, которые могут оказаться полезными, если когда-нибудь вы решите, что разработка программного обеспечения – это то, чем вы хотите заниматься. Другой путь научиться разрабатывать программы – послать свое резюме на *jobs@fogcreek.com* и подать заявление на летнюю практику. Кое-чему мы вас там научим.

Большое спасибо за внимание.

ГЛАВА ДЕСЯТАЯ

Советы студентам, изучающим вычислительную науку

2 января 2005 года, воскресенье

Несмотря на то что лишь год или два назад я распространялся по поводу того, что богатые графические клиенты Windows – наше будущее, тем не менее я иногда получаю письма от студентов с просьбой посоветовать им, в каком направлении продвигать свою карьеру. А поскольку сейчас как раз сезон приема на работу, я решил изложить свои обычные рекомендации, которые можно прочесть, посмеяться и забыть.

К счастью, большинство студентов достаточно самонадеянны и никогда не обращаются за советом к старшим, что в области вычислительной науки весьма разумно, потому что старшие склонны говорить допотопные глупости вроде «спрос на набивщиков перфокарт к 2010 году превысит 100 000 000» или «сейчас везде требуются программисты на Lisp».

Я тоже понятия не имею, о чем говорю, когда даю советы студентам. Я так безнадежно отстал, что не могу разобраться с AIM и по старинке пользуюсь (вот ужас!) электронной почтой, популярной в те времена, когда музыка выходила на плоских круглых пластинках, называемых «CD».

Поэтому лучше не слушать то, что я здесь говорю, а вместо этого написать какую-нибудь сетевую программу, помогающую студентам назначать свидания.

И все же.

Если вам нравится программировать, то у вас есть ряд преимуществ. Во-первых, вы из тех немногих счастливчиков, кто может неплохо зарабатывать себе на жизнь, занимаясь тем, что им нравится. Большинству повезло меньше. Сама идея «любить свою работу» достаточно нова. Всегда счи-

талось, что работа – нечто неприятное, чем нужно заниматься, чтобы заработать деньги, которые нужны для того, чтобы, выйдя в 65 лет на пенсию, вы наконец-то смогли заняться тем, что вам нравится, – если хватит денег, и вы еще не слишком дряхлы для этого дела в случае, когда оно требует крепких ног, хорошего зрения, способности пройти пять метров без одышки и так далее.

Так, о чем я говорил? Ах, да, советы.

Чтобы больше не отвлекаться, вот вам Семь Бесплатных Советов Джоэла Студентам, изучающим вычислительную науку (стоят они ровно столько, сколько вы за них заплатили):

1. Научитесь писать до окончания учебы.
2. Изучите язык C до окончания учебы.
3. Изучите микроэкономику до окончания учебы.
4. Не пренебрегайте курсами, не имеющими непосредственного отношения к вычислительной науке, даже если они кажутся вам неинтересными.
5. Выберите курсы, где активно занимаются программированием.
6. Не бойтесь, что все рабочие места для программистов окажутся в Индии.
7. Постарайтесь найти хорошую летнюю практику.

А теперь читайте обоснования, если только вы не настолько доверчивы, чтобы бездумно следовать моим советам, ибо тогда добавится еще один пункт, восьмой: найдите специалиста, который поможет вам поднять самооценку.

Научитесь писать до окончания учебы

Достигла бы Linux такого успеха, если бы Линус Торвальдс не сумел так успешно ее пропагандировать? Да, он блестящий программист, но именно благодаря его таланту излагать свои мысли по-английски и распространять их с помощью электронной почты и почтовых списков Linux привлекла множество добровольных разработчиков со всех концов света.

Вы слышали о последнем веянии моды – «экстремальном программировании»? Не стану высказывать здесь свое мнение об XP, но причина, по которой вам стало известно о нем, это его рекламирование очень одаренными писателями и ораторами.

Даже в ограниченных рамках какой-нибудь организации, занимающейся программированием, наибольшей властью и влиянием пользуются те программисты, которые пишут и говорят по-английски понятно, убедительно и спокойно. Иметь высокий рост тоже полезно, но тут что-либо изменить уже не в ваших силах.

Разница между неплохим и выдающимся программистами не в количестве языков программирования, которыми они владеют, и не в том, какой язык предпочитают, – Python или Java. Разница в их способности доносить свои идеи. Их сила – в умении убеждать других. Они пишут ясные комментарии и технические спецификации, благодаря чему их код понятен другим программистам, а значит, другие программисты могут использовать и развивать их код, вместо того чтобы переписывать его. Без этого код бесполезен. Благодаря написанной ими четкой технической документации для конечных пользователей те смогут понять, какую работу выполняет этот код, и это единственный способ для таких пользователей увидеть ценность этого кода. Где-то в глубинах SourceForge погребена масса прекрасного и полезного кода, которым никто не пользуется, потому что создавшие его программисты – плохие писатели (или вообще никакие), и никто не знает об их трудах, и их великолепный код так и чахнет где-то.

Я не беру на работу программистов, которые не умеют писать, – то есть хорошо писать – по-английски. Если вы умеете писать, то, где бы вы ни работали, вам скоро предложат написать спецификацию, и это будет означать, что вы уже приобрели некоторый вес и замечены руководством.

В большинстве колледжей есть курсы, обозначенные как «требующие интенсивной письменной работы», то есть для сдачи экзаменов по ним нужно исписать массу страниц. Найдите такой курс и запишитесь на него! Ищите курсы из любой области, где есть еженедельные или ежедневные письменные задания.

Заведите себе дневник или блог. Чем больше вы станете писать, тем лучше научитесь это делать, и наоборот: чем лучше вы будете это делать, тем больше станете писать.

Изучите язык C до окончания учебы

Пункт второй: язык C. Обратите внимание: я не сказал «C++». Хотя C встречается все реже, это по-прежнему лингва-франка профессиональных программистов. Это язык, на котором они общаются между со-

бой, и, главное, он гораздо теснее связан с машиной, чем «современные» языки, которым вас учат в колледжах, вроде ML, Java, Python или еще какой-нибудь модной сегодня ерунды. Нужно хотя бы в течение семестра тесно поработать с машиной, иначе вы никогда не научитесь писать эффективный код на языках верхнего уровня. Вы также никогда не сможете заниматься компиляторами или операционными системами, а этот вид работы – один из лучших для программиста. Вам никогда не поручат разрабатывать архитектуру крупного проекта. Мне все равно, в какой мере вы знакомы с продолжениями, замыканиями и обработкой исключений: если вы не можете объяснить, почему `while (*s++ = *t++);` копирует строку, и это не самая очевидная для вас вещь на свете, значит, вы, по моим представлениям, программируете безрассудно – как врач, который, не зная основ анатомии, выписывает рецепты в соответствии с рекомендациями рекламного агента фармацевтической фирмы.

Изучите микроэкономику до окончания учебы

Очень краткое замечание для тех, кто не прослушал ни одного курса по экономике: эти курсы из тех, которые очень хорошо начинаются, излагая полезные теории и осмысленные факты с практическими примерами и так далее, а потом их уровень неуклонно снижается. То, что было полезно в начале, это микроэкономика, служащая основой буквально каждой значительной теории в бизнесе. Затем наступает деградация: вы переходите к макроэкономике (которую вполне можно опустить) с ее интересными теориями вроде связи процентных ставок и безработицы, для чего есть больше опровержений, чем доказательств, а дальше все становится еще хуже, и даже те, кто выбрал основной специальностью экономику, часто переходят на физику, что, кстати, помогает им лучше устроиться на Уолл-стрит. Но микроэкономику следует изучить, потому что нужно понимать, что такое спрос и предложение, что такое конкурентное преимущество и чистая приведенная стоимость, дисконтирование и предельная полезность, чтобы получить представление о том, как действует бизнес.

Зачем изучать экономику тому, кто выбрал основной специальностью вычислительную науку? Потому что программист, разбирающийся в основах бизнеса, ценнее для бизнеса, чем тот, который не разбирается. Только и всего. Меня многократно приводили в отчаяние программисты, выдвигавшие сумасшедшие идеи, которые имели смысл в коде – и никакого смысла в капитализме. Если вы разбираетесь в этих вещах, то ваша цен-

ность как программиста возрастает, и ваше вознаграждение тоже, о причинах чего также можно узнать, изучая микроэкономику.

Не пренебрегайте курсами, не имеющими непосредственного отношения к вычислительной науке, даже если они кажутся вам неинтересными

Пренебрежение курсами, не относящимися к специальности, – верный способ снизить свой средний балл (GPA). Отнеситесь к своему GPA со всей серьезностью. Многочисленные кадровые агенты и работодатели – и я в их числе – сразу ищут в резюме GPA, и в этом нет ничего предосудительного. В конце концов, GPA лучше, чем любой другой числовой показатель, отражает мнение десятков преподавателей о вашей работе, сложившееся на протяжении долгого времени и в многочисленных ситуациях. Результат экзамена SAT? Это всего лишь один тест, длящийся несколько часов. В GPA нашли отражение сотни контрольных работ, промежуточных экзаменов и работы в классе на протяжении четырех лет. Да, с ним связаны свои проблемы. На протяжении ряда лет происходила инфляция этой оценки. Ваш GPA ничего не говорит о том, как вы его получили, – через простые курсы экономики домашнего хозяйства в коммунальном колледже или изучая квантовую механику на старших курсах Калифорнийского технологического института. Отсеяв всех с GPA 2.5 из коммунального колледжа, я требую характеристики и рекомендации. А затем смотрю, чтобы высокие оценки были *по всем* предметам, а не только по вычислительной науке.

Почему мне, работодателю, который ищет программистов, интересно, какие оценки у кандидата по европейской истории? В конце концов, история – скучная вещь. Вы говорите, что я должен взять вас, хотя когда работа скучная, вы не очень стараетесь? В программировании тоже есть скучные вещи. Любая работа временами бывает скучной. И я не хочу принимать на работу людей, которые хотят заниматься только интересными вещами.

Во время учебы я записался на курс под названием «культурная антропология», потому что решил – должен же я *что-то* знать об антропологии, а этот курс показался мне интересным обзором.

Интересным? Если бы я только знал, что меня ждет! Мне пришлось читать страшно нудные книги об индейцах влажных лесов Бразилии и аборигенах островов Тробриан, которые, при всем уважении к ним, меня во-

все не интересуют. В какой-то момент занятия стали такими нудными, что я предпочел бы им даже такое захватывающее зрелище, как *наблюдение за ростом травы*. Я совершенно утратил интерес к этому предмету. Полностью и окончательно. Слезы текли из глаз – так я устал от бесконечных рассказов о сборе картофеля. Не знаю, почему тробрианские островитяне столько времени посвящали сбору картофеля, не могу вспомнить – это очень скучно, но по этому курсу предстоял экзамен в середине семестра, поэтому я упрямо корпел над ним. В конечном итоге, я решил, что культурная антропология – это данное мне судьбой испытание готовности противостоять скуке. Если я смогу получить «А» по курсу, который требует от меня заучить все, что касается индейского праздника с приношением даров, то смогу выдержать любое дело, каким бы скучным оно ни оказалось. Когда впоследствии я случайно попал в Линкольнский центр и должен был просидеть восемнадцать часов, слушая «Кольцо Нибелунгов» Вагнера, это показалось мне едва ли не удовольствием по сравнению с изучением обычаев народа Квакиутл, которое я должен был вспомнить с благодарностью.

Я получил свою «А». И если я смог это сделать, то и вы сможете.

Выберите курсы, где активно занимаются программированием

Я точно помню, когда я решил не поступать в аспирантуру. Это было во время курса динамической логики, который в Йеле читала Ленора Зак, блестящий преподаватель факультета CS.

Едва ли мои мрачные воспоминания помогут воздать должное этой области, но все же позвольте довести рассказ до конца. Идея формальной логики состоит в том, что вы доказываете справедливость одних вещей, исходя из справедливости других. Например, в силу формальной логики предложения «всех, у кого хорошие оценки, берут на работу» и «у Джонни хорошие оценки» в сумме позволяют вывести новую истину, а именно, что «Джонни возьмут на работу». Это все довольно причудливо, и деконструктивисту хватит десяти секунд, чтобы расправиться со всем полезным, что есть в формальной логике, оставив вам нечто забавное, но бесполезное.

Динамическая логика – это то же самое, но дополнительно появляется время. Например, из «*После* того, как свет включен, вы можете видеть свои ботинки» плюс «Свет включен ранее» следует «Вы можете видеть свои ботинки».

Динамическая логика привлекает таких блестящих теоретиков, как профессор Зак, потому что она укрепляет надежду на *формальное* доказательство предложений, касающихся компьютерных программ, что могло бы принести пользу в случае, скажем, *формального* доказательства того, что память на флеш-карте марсохода не переполнится, и он не станет непрерывно перезагружаться вместо того, чтобы ездить по Красной планете и разыскивать на ней марсиан.

Итак, на первой лекции курса профессор Зак исписала две классные доски и изрядную часть соседней стены доказательством того, что если у вас есть электрический выключатель, и свет не горит, то, щелкнув выключателем, вы зажжете свет.

Доказательство было безумно сложным, и допустить в нем ошибки было очень легко. Труднее было проверить корректность *доказательства*, чем поверить, что, щелкнув выключателем, вы зажжете свет. На самом деле, во всем этом длиннющем доказательстве было опущено множество переходов, потому что формально обосновывать их было бы слишком скучно. Одни переходы обосновывались излюбленным методом доказательства по индукции, другие доказывались от противного, были еще такие, справедливость которых подтверждалась одним из старшекурсников.

На дом было задано доказать обратное: *если* света не было *и* теперь он горит, значит, вы щелкнули выключателем.

Я честно пытался доказать это.

Я долго просидел в библиотеке.

Через пару часов я обнаружил ошибку в первоначальном доказательстве, которое я пытался смоделировать. Может быть, я просто ошибся, когда записывал его, но это привело меня к некоторому выводу: если нужно три часа, чтобы записать на доске доказательство тривиального факта, и при этом сто раз можно допустить ошибку, то таким методом никогда не удастся доказать что-либо *интересное*.

Но для тех, кто занимается математической логикой, это не имеет никакого значения: им интересна не *польза*, а сам процесс.

Я бросил этот курс и поклялся никогда не поступать в аспирантуру по вычислительной науке.

Вывод из всей этой истории: вычислительная наука и разработка программного обеспечения – разные вещи. Если вам очень повезет, то в учебном плане вашего колледжа разработка программного обеспечения может занимать приличное место, но так бывает не всегда – в элитных учебных

заведениях полагают, что практические навыки лучше получать в профессионально-технических училищах и по программам реабилитации бывших заключенных. Обычное *программирование* можете изучать в каком-то другом месте. Мы – Йельский университет, мы Штампуем Будущих Мировых Лидеров. Думаете, вы платите 160 000 долларов за право изучать циклы while? Что вам тут – *семинар по Java* в отеле Airport Marriott?

Тьфу!

Беда в том, что у нас фактически нет профессионального обучения разработке программного обеспечения, поэтому если вы решили стать программистом, то, скорее всего, выберете в качестве основной специальности вычислительную науку. Это замечательная специальность, но разработка программного обеспечения – несколько *иное дело*.

Однако вам может повезти, и на факультете CS окажется много курсов, где интенсивно занимаются программированием, так же как на историческом факультете можно найти достаточно курсов, где придется столько писать, что вы научитесь это делать. Это лучшие курсы, какие можно выбрать. Если вы любите программирование, не отчаивайтесь, что вам не понятен смысл курсов по лямбда-исчислению или линейной алгебре, где вам не придется притронуться к компьютеру. Ищите курсы уровня 400 со словом «практикум» в названии. Это просто способ скрыть полезный курс от либерально-претенциозной администрации с помощью латинского названия.

Не бойтесь, что все рабочие места для программистов отдадут в Индию

Прежде всего, если вы живете *в* Индии, вам нечего было беспокоиться по этому поводу раньше и нечего *начинать* тревожиться сейчас о тех рабочих местах, которые уходят в Индию. Это замечательные рабочие места, дай бог вам здоровья.

Но мне постоянно приходится слышать об опасном снижении числа поступающих на факультеты CS, одна из причин которого, якобы, «боязнь студентов связываться с областью, в которой все работы уходят в Индию». Это совершенная неправда, и вот почему. Во-первых, глупо выбирать себе карьеру, основываясь на сегодняшней моде в бизнесе. Во-вторых, изучение программирования дает очень хорошую подготовку для целого ряда замечательно интересных областей, таких как проектирование бизнес-процессов, и пусть хоть все связанные с программированием работы

уйдут в Индию и Китай. В третьих, – можете мне поверить – настоящих программистов очень не хватает – как здесь, так *и* в Индии. Да, есть группа безработных ИТ-специалистов, которые подняли большой шум по поводу того, что долго не могут найти себе работу. Но знаете, что я вам скажу? Может, им это и не понравится, но по-настоящему хороший программист *найдет* себе работу. В-четвертых, у вас есть на примете что-то лучшее? Вы собираетесь выбрать специальностью историю? Тогда вам остается только юридический колледж. Но мне доподлинно известно: 99% практикующих юристов *ненавидят* свою работу; кроме того, они работают по 90 часов в неделю. Я уже говорил: если вы любите программировать, вам сильно повезло, потому что вы окажетесь среди тех немногих, кто хорошо зарабатывает, занимаясь любимым делом.

Но мне кажется, что студенты об этом не задумываются. Сокращение числа поступающих на факультеты CS – это просто восстановление исторически оправданного уровня после неумеренного роста во время дотком-бума. Тот, прошлый пузырь, раздули те, кто, не особенно любя программирование, считал, что факультет CS поможет им получить высокооплачиваемую работу и вывести собственную компанию на биржу в 24 года. К счастью, таких уже давно нет.

Постарайтесь найти хорошую летнюю практику

Умные охотники за головами знают, что тот, кто любит программирование, в восьмом классе написал базу данных для своего зубного врача и три года вел занятия в летнем компьютерном лагере, прежде чем поступить в колледж, а потом построил систему управления контентом для газеты своего кампуса и проходил летнюю практику в софтверных компаниях. Вот такие вещи они и ищут в ваших резюме.

Если вам нравится программирование, то самая большая ошибка, которую вы можете совершить, это наняться – летом, на полставки или еще как – на какую-то работу, не связанную с программированием. Я знаю, что очень многие 19-летние хотят подработать в универмаге, складывая рубашки, но у вас есть очень ценные умения, даже если вам всего 19, и глупо складывать рубашки вместо того, чтобы воспользоваться этими навыками. К моменту окончания учебы в вашем резюме должна быть перечислена целая куча программистских работ. Выпускники Abercrombie & Fitch[1] в буду-

[1] «Аберкромби энд Фитч» – сеть популярных в Америке магазинов.

щем поступают на работу в Enterprise Rent-a-Car[1], «помогая людям решать их проблемы с арендой автомобилей». (Кроме Тома Веллинга. Он играет Супермена на ТВ.)

Чтобы значительно облегчить вам жизнь и подчеркнуть, что вся эта статья посвящена исключительно заботе о наших собственных интересах, сообщаю, что моя компания Fog Creek Software принимает студентов на летнюю практику по программированию, прохождение которой сильно повысит ценность вашего резюме. «Вы, несомненно, больше узнаете о написании кода, разработках и бизнесе во время практики в Fog Creek Software, чем в каком-либо другом месте», – пишет Бен, один из наших прошлогодних практикантов, и я не подсылал в общежитие наемного бандита, чтобы заставить его это признать. Заявления принимаются до 1 февраля. Спешите.

Если вы послушаетесь моих советов, это может также кончиться тем, что вы поспешно продадите свои акции Microsoft и откажетесь от работы в Google, потому что захотите иметь собственный офис с отдельным входом, или совершите еще какие-нибудь глупости, но я в этом не виноват. Я же предупреждал: не слушайте меня.

[1] Крупнейшая в Америке компания по аренде транспортных средств.

ЧАСТЬ ТРЕТЬЯ

8 Значение проектирования

ГЛАВА ОДИННАДЦАТАЯ

Сглаживание шрифтов, антиалиасинг и субпиксельная прорисовка

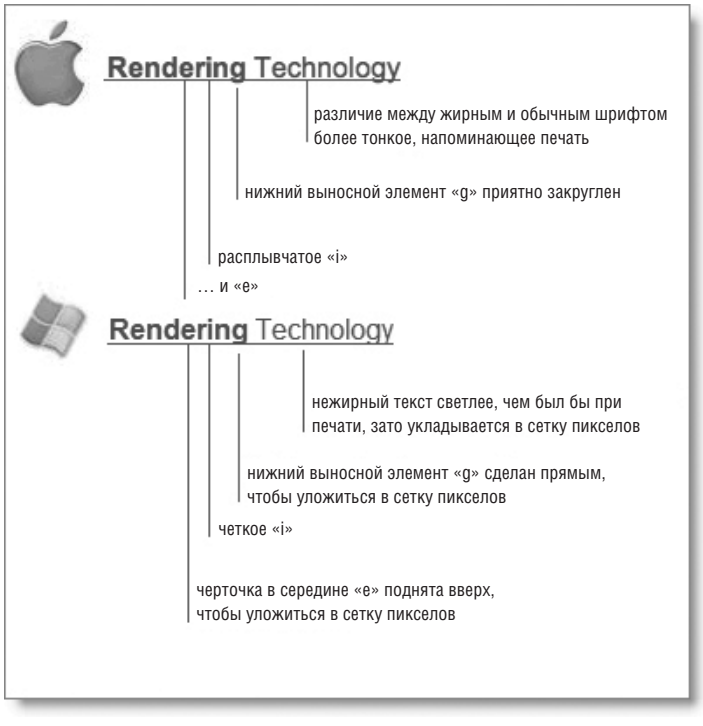
12 ИЮНЯ 2007 ГОДА, ВТОРНИК

Apple и Microsoft всегда по-разному решали проблему отображения шрифтов на экране компьютера. Сегодня обе компании применяют субпиксельную прорисовку, или субпиксельный рендеринг, чтобы шрифты на типичных экранах с низким разрешением были более четкими. Но при этом у них разный подход.

- В Apple считают, что алгоритмы должны сохранять рисунок гарнитуры, даже ценой некоторой расплывчатости.
- В Microsoft считают, что границы каждого символа нужно соблюсти до пиксела, чтобы избежать расплывчатости и улучшить читаемость, даже ценой отхода от рисунка гарнитуры.

С появлением броузера Safari для Windows, в котором старательно применяются алгоритмы прорисовки Apple, вы можете сами сравнить эти два подхода у себя на экране (см. рисунок) и понять, что я имею в виду. Полагаю, вы заметите разницу. Шрифты Apple действительно нечеткие, с расплывчатыми краями; но когда размер шрифта невелик, разные семейства шрифтов заметно отличаются между собой, потому что прорисовываются ближе к тому, как они выглядели бы при печати с высоким разрешением. (Эта картинка есть на странице *http://www.joelonsoftware.com/items/2007/06/12.html*.)

Источник различия – прежний опыт Apple в настольной печати и графическом дизайне. Удобное свойство алгоритма Apple – возможность сделать макет страницы для печати и увидеть на экране хорошее приближе-



ние к конечному результату. Это особенно важно для оценки плотности блока текста. Применяемый в Microsoft механизм подгонки шрифта по пикселам приводит к появлению слишком тонких линий, чтобы избежать расплывчатости, и в результате абзац может выглядеть более светлым, чем он окажется на печати.

Преимущество метода Microsoft в том, что он облегчает чтение с экрана. Microsoft прагматично решила, что не стоит слишком благоговеть перед начертанием шрифта, и резкость шрифта на экране, облегчающая чтение, важнее представлений создателя шрифта о том, насколько светлым или темным должен выглядеть целый абзац текста. На самом деле, Microsoft даже разработала специальные экранные шрифты, такие как Georgia и Verdana, привязанные к границам пикселов. Они отлично выглядят на экране, но не на печатном листе.

Apple предпочла элегантность практичности, потому что у Стива Джобза есть вкус, тогда как Microsoft пошла удобным путем, выбрав прагматич-

ный подход, совершенно чуждый щегольству. Иными словами, Apple отличается от Microsoft тем же, чем Target от Wal-Mart.

Вопрос в том, какой из подходов больше нравится пользователям. Пост Джеффа Этвуда (Jeff Atwood, *www.codinghorror.com/blog/archives/000884.html*), в котором он сопоставил обе эти технологии, вызвал вполне предсказуемые разногласия: пользователям Apple больше нравится система Apple, а пользователям Windows – система Microsoft. Это не обычный фанатизм; здесь нашел отражение тот факт, что если предлагается выбрать лучший стиль или конструкцию, неподготовленный человек скорее выберет то, что ему привычнее. При исследовании вкусов и предпочтений люди часто уходят от выбора, останавливаясь на том, что выглядит знакомо. Это касается чего угодно – от столового серебра (выбирают похожее на то, которое было у их родителей) до гарнитуры шрифта и графического дизайна: если предварительно не объяснить людям, к чему нужно присмотреться, они выберут то, что им лучше всего знакомо.

Возможно, именно поэтому инженеры Apple считают, что оказывают сообществу Windows большую услугу, предоставляя этим варварам свою «более высокую» технологию прорисовки шрифтов, и именно поэтому пользователи Windows обычно считают, не зная истинных причин, что Safari передает шрифты расплывчатым и странным образом, и просто не любят этот броузер. На самом деле, вот что они думают: «Опа! Что-то не так. А я не люблю, когда что-то не так. Почему мне не нравятся эти шрифты? Ага, вблизи они кажутся расплывчатыми! *Наверно, потому и не нравятся*».

ГЛАВА ДВЕНАДЦАТАЯ

Счет на дюймы

7 ИЮНЯ 2007 ГОДА, ЧЕТВЕРГ

– Кто-то не выключил радио в ванной? – спросил я Джареда. Играла тихая классическая музыка.

– Нет, это снаружи. Это началось, пока ты был в отъезде, и теперь происходит каждую ночь.

Мы живем в Нью-Йорке, в многоквартирном доме. Со всех сторон нас окружают соседи. Мы уже привыкли к тому, что снизу смотрят ТВ-шоу, а у ребенка в квартире над нами есть любимая игра: бросить на пол горсть шариков, а потом броситься за ними *самому*. Я пишу эти строки, а в это время он носится по комнатам, круша мебель. Скорей бы он повзрослел и занялся пейнтболом.

Однако классической музыки по ночам раньше не бывало.

Что еще хуже, это была какая-то романтическая музыка в стиле «Буря и натиск», и я впал в неистовство, вместо того чтобы провалиться в сон.

Наконец, музыка смолкла, и я смог уснуть. Но в следующую полночь музыка заиграла вновь, доводя меня до изнурения, причем *опять* звучала какая-то ерунда самодовольного Вагнера с помпезными крещендо, будившими меня, стоило только задремать, и мне пришлось разглядывать кошачьи фотографии, сидя в гостиной, пока около часа ночи это наконец не прекратилось.

На следующую ночь мое терпение лопнуло. Когда около полуночи послышалась музыка, я оделся и начал обследовать здание. Я крался по коридорам, прислушиваясь у каждой двери и пытаясь определить, откуда доно-

сятся звуки. Я высовывался из окон и нашел незапертую дверь в вентиляционную шахту, где музыка была заметно громче. Я поднимался и спускался по лестнице, прислушиваясь к окну на каждой площадке, пока не убедился, что источником проблемы была квартира милой пожилой миссис С – прямо под нами, № 2В.

Я не поверил, что миссис С в свои шестьдесят бодрствует в столь поздний час или спит под такую громкую музыку, разве что она нарочно задержалась, чтобы послушать передачу из какого-нибудь цикла «Кольцо Нибелунгов» на канале классической музыки.

Как оказалось, ничего подобного.

Я заметил, что музыка включалась каждую полночь и выключалась ровно в час ночи. Это навело меня на мысль о радиобудильнике с заводской настройкой на срабатывание в 00:00.

Не смея разбудить старушку из-за одного лишь *подозрения*, что музыка доносится из ее квартиры, в отчаянии я вернулся к себе и занялся разглядыванием *xkcd*. Я был подавлен и зол, потому что не смог решить проблему. Я мучился и злился весь следующий день.

Вечером я постучал к миссис С. Управляющий сказал, что через день она уезжает на все лето, поэтому если проблемы связаны с ее квартирой, лучше разобраться с ними безотлагательно.

– Извините за беспокойство, – сказал я, – дело в том, что каждый раз около полуночи из вентиляционной шахты за нашими квартирами доносится громкая классическая музыка, и это мешает мне заснуть.

– О нет, это не я, – возразила она, как и ожидалось. Конечно, это не ее вина: наверняка она ложится спать вовремя и никогда не включает громкую музыку, чтобы не беспокоить соседей.

Я высказал предположение, что она туговата на ухо и может просто не заметить, что в соседней комнате посреди ночи завывает эта штука. Или что у нее слишком крепкий сон.

Мне пришлось потратить на уговоры несколько минут, прежде чем она все же согласилась посмотреть, нет ли какого-нибудь радиобудильника в той комнате, что подо мной.

Он был там. Прямо на окне, под окном моей спальни. Увидев, что он настроен на самый классический из музыкальных каналов, я понял, что виновник найден.

– А, это? Понятия не имею, как им пользоваться. Даже не трогаю его, – сказала она. – Выключу его совсем.

– Не нужно, – сказал я, отключил будильник, вырубил звук и, следуя синдрому навязчивых состояний, выставил точное время.

Миссис С умоляла простить ее, но, в сущности, она была не виновата. Даже я (я!) далеко не сразу понял, как управлять этим чертовым будильником, – а ведь я кое-что знаю о *радиобудильниках*, не сомневайтесь. Интерфейс пользователя был ужасен. У бедной старушки не было ни единого шанса.

Значит, виноват радиобудильник? В какой-то мере. Им было слишком сложно пользоваться. Там был будильник, который срабатывал каждые сутки, даже если весь день его никто не трогал, – не самое лучшее решение. После отключения питания этот сигнал устанавливался на *полночь*, хотя семь утра – более разумный выбор для настройки по умолчанию.

Почему-то последние несколько недель я *ко всему* отношусь чересчур критически. Я всюду ищу недостатки, а найдя, упорно думаю, как их исправить. Это особое состояние ума, свойственное всем программистам на этапе окончательной отладки нового продукта.

Последние несколько недель я дописываю документацию для очередной важной версии FogBugz. Описывая что-нибудь, я проверяю, действительно ли это работает так, как полагается, или делаю снимки экрана. И чуть ли не каждый час раздается сигнал тревоги: «Да что же это? Все должно работать не так!»

Поскольку речь о программе, я всегда могу ее исправить. ХА-ХА! Шутка! Я уже не смогу разобраться в этом коде. Я просто регистрирую ошибку, а исправит ее тот, кто догадается, что нужно сделать.

Дэйв Вайнер (Dave Winer) пишет: «Чтобы создать пригодную для использования программу, нужно бороться за каждое исправление, каждую функцию, каждое мелкое удобство, если это облегчит жизнь хотя бы одному пользователю. Кратчайших путей здесь нет. Удача имеет значение, но победа зависит не от нее, а от упорной борьбы за каждый дюйм» (*www.scripting.com/2002/01/12.html*).

В коммерческих программах – тех, которые вы продаете другим людям, – счет идет на дюймы.

Каждый день вы делаете маленький шаг вперед. Какая-то функция заработала чуть-чуть лучше. Ваш будильник устанавливается по умолчанию на семь утра вместо полуночи. Маленькое усовершенствование, которое мало кому пригодится. Один дюйм.

И таких мелочей тысячи.

Чтобы находить их, нужен особый, критический склад ума. Вы должны преобразовать свой ум, так чтобы замечать *любые* недостатки. У ваших близких едет крыша. В семье хотят вас убить. Если, идя пешком на работу, вы видите тупой водительский маневр, только собранная в кулак воля не даст вам пойти и объяснить этому водителю, что он чуть не убил вон того бедного ребенка в инвалидном кресле.

И пока вы исправляете одну мелкую деталь за другой, шлифуете, доводите, полируете и *отделываете* все укромные уголки своего продукта, происходит нечто удивительное. Дюймы складываются в футы, футы – в ярды, а ярды – в мили. И в результате появляется действительно прекрасный продукт. Продукт, который замечательно выглядит, интуитивно понятно работает и от которого люди в восторге. И когда один из миллиона пользователей этого продукта попытается сделать то, что не придет в голову прочим, он обнаружит, что это не только возможно, но и работает великолепно: даже в кладовках вашей программы мраморные полы, дубовые двери и стены, обшитые панелями полированного красного дерева.

И это признак замечательного программного продукта.

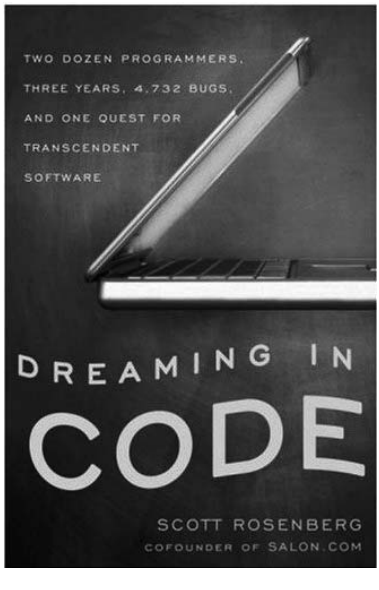
Поздравляю команду FogBugz 6.0, умеющую с удивительным успехом вести борьбу за дюймы, выпустившую сегодня свою первую бета-версию и готовящуюся к выпуску окончательной версии в конце лета. Это лучший из когда-либо созданных ею продуктов. Он приведет вас в восторг.

ГЛАВА ТРИНАДЦАТАЯ

Общая картина

21 января 2007 года, воскресенье

Рецензия на книгу «Dreaming in Code» (Грезы в коде), Scott Rosenberg (Скотт Розенберг), Three Rivers Press, 2007.



В зрении участвует механизм обращения к отсутствующей странице. Настолько успешно, что вы даже не замечаете этого.

Видеть с высокой степенью разрешения вы можете только в относительно небольшой области, и даже там, в самом ее центре, есть крупное пятно, в котором видимости нет, и несмотря на это, вы *полагаете*, что у вас панорамное зрение со сверхвысоким разрешением. Почему? Потому что ваши глаза очень быстро двигаются, и в обычной ситуации они мгновенно перенаправляются туда, куда вам требуется. А ваш мозг поддерживает это совершенно абстрактное представление, создавая иллюзию полного обзора, тогда как *на самом деле* вы располагаете очень маленькой областью острого зрения, обширной областью очень слабого зрения и способностью загружать отсутствующую страницу памяти для всего, что хотите видеть, – так быстро, что можно не сомневаться: этот маленький театр в вашем мозгу показывает всю картину.

Это чрезвычайно полезная схема, применяемая и во многих других ситуациях. Ваш слух способен улавливать важные части разговора. Ваши пальцы шарят повсюду и ощупывают все, что нужно, – шерстяной свитер или внутренность носа, – давая полную картину ощущений. Во время сна ваш мозг задает органам чувств те же вопросы, что и наяву (взгляни-ка, что там?!), но органы чувств временно отключены (вы же *спите*, в конце концов), и мозг, подставляя ответы как попало, сплетает причудливые рассказы, называемые снами. А утром, пытаясь пересказать приятелю свой сон, несмотря на *кажущееся* ощущение абсолютной реальности происходившего, вы вдруг обнаруживаете, что *на самом деле ничего не помните*, и сочиняете что-то на ходу. За следующую минуту-другую сна ваш мозг успел бы спросить у органов чувств, *что* это за млекопитающее плавает вместе с вами в розовом кусте, и получить любой дурацкий ответ (*утконос*!), но вы проснулись – и даже не осознавали отсутствие потребности определить, что было рядом с вами в розовом кусте, до попытки связно передать свой сон приятелю. Что у вас никогда не получится. Так что прошу не рассказывать мне свои сны.

Один из неприятных побочных эффектов состоит в том, что у мозга появляется скверная привычка переоценивать точность своего понимания вещей. Начинает казаться, что он *всегда* владеет полной картиной, даже когда это не так.

Особенно опасна эта ловушка при разработке программного обеспечения. У вас в голове есть некоторая общая картина того, что вы хотите сделать, и все выглядит настолько *кристально ясным*, что кажется, будто и *проектировать* ничего не нужно. Просто садись и реализуй свою мечту.

Допустим, к примеру, что эта ваша мечта состоит в том, чтобы переделать старую программу-ежедневник для DOS, *очень-очень хорошую*, но *совершенно недооцененную*. Кажется, что это несложно. Ее работа выглядит настолько очевидной, что вы даже не пытаетесь проектировать новый продукт, а просто набираете команду программистов, которые тут же начинают выдавать код.

При этом вы сделали две ошибки.

Во-первых, вы попали в ту самую старую ловушку, когда ваш мозг переоценивает свои силы. «Да у нас нет никаких сомнений в том, как это сделать! Все абсолютно ясно. Не нужно писать спецификацию. Сразу пишем код».

Во-вторых, вы набрали программистов раньше, чем спроектировали продукт. Потому что труднее проектирования программного продукта может быть только попытка проектировать его целой *командой*.

Я миллион раз видел, как обсуждение работы некой части программы заходит в тупик, если в нем участвует хотя бы один или два других программиста. В результате я иду к себе в кабинет, беру лист бумаги и начинаю чертить. Сама необходимость с кем-то общаться мешает мне хорошо сосредоточиться и придумать, как должна работать эта чертова функция.

Что меня убивает, так это дурная привычка некоторых команд собирать *совещание*, чтобы выяснить, как должна работать какая-то функция. Вы не пробовали коллективно сочинять стихи? Представьте банду тупых каменщиков, которые, сидя на диване, пытаются одновременно смотреть «Спасателей Малибу» и писать оперу. Чем больше тупых каменщиков на диване, тем меньше шансов, что у них получится опера.

Хоть телевизор-то выключите!

Итак, я даже не пытаюсь угадать, что произошло с командой Chandler, и зачем они потратили несколько лет и миллионы долларов на то, чтобы оказаться в нынешнем положении, то есть выпустить полное ошибок и недоработок приложение-календарь, немногим лучше десятков появившихся в последнее время календарей-близнецов в стиле Web 2.0, каждый из которых создан парой студентов на досуге, причем наверняка второй понадобился только затем, чтоб рисовать талисманы.

У Chandler нет даже талисмана!

Повторяю, я не берусь утверждать, с какими неприятностями они столкнулись. Может быть, ни с какими. Может быть, они считают, что все идет по плану. Отличная книга Скотта Розенберга (Scott Rosenberg), пред-

полагаемая «Soul of a New Machine» («Душа новой машины») самого замечательного open source-проекта десятилетия, заканчивается на печальной ноте. Скотт оборвал повествование, потому что было ясно – ждать скорого выпуска Chandler 1.0 не приходится (может быть, Розенберг опасался, что ко времени выхода программы книг вообще уже не будет, а все знания мы станем получать, глотая таблетки).

Тем не менее, это замечательный обзор одного из конкретных типов программных проектов – тех, которые непрерывно движутся, но никуда не прибывают, потому что замыслы были слишком величественны, а деталей немного не хватало. Насколько я могу судить, первоначальное видение Chandler в значительной мере ограничивалось тем, что продукт должен быть «революционным». Не знаю, как вы, но я не умею писать «революционный» код. Чтобы написать код, мне нужны кое-какие подробности. Если в спецификации продукт описывается с помощью эпитетов («должно быть суперкруто»), а не конкретных характеристик («панели заголовков должны быть в стиле необработанного алюминия, а у значков должно быть легкое отражение, как будто они стоят на крышке рояля»), готовьтесь к неприятностям.

Насколько я понял из книги Розенберга, единственными конкретными идеями проекта были «одноранговость», «совместное хранение данных» и «интерпретация дат на естественном языке». Возможно, такая ограниченность присутствует только в книге, но в том, что проект изначально был крайне туманным, нет никаких сомнений.

«Одноранговость» была raison d'être[1] для Chandler – стоит ли покупать Microsoft Exchange Server чтобы координировать графики? Выяснилось, что одноранговая синхронизация слишком сложна или связана с какими-то проблемами, поэтому данную функцию убрали. Появился сервер под именем Cosmo.

«Совместное хранение данных» предполагало, что вместо того, чтобы хранить почту в одной куче, календарные события в другой, а заметки в третьей, нужно сделать общую кучу, в которой будет храниться все.

Размышляя над этой идей, приходишь к выводу, что она неудачна. Размещать электронную почту в календаре? Где? В день получения? Чтобы в результате я, получив в пятницу 200 писем с рекламой виагры, не заметил важную встречу – собрание акционеров?

---

[1] Смысл существования, разумное основание (фр.). – *Прим. перев.*

В результате «совместное хранение данных» было переработано в идею *марок*, чтобы, например, можно было «наклеить почтовую марку» на любой документ, заметку или календарное событие и послать их всем, кому нужно. Сообщаю: такая функция существовала в Microsoft Office лет десять. Из Office 2007 ее наконец-то *изъяли*, потому что она была никому не нужна. И без того есть масса простых способов отправить что-то по почте.

Мне кажется, что идеей «совместного хранения данных» могут соблазниться астронавты от архитектуры, любители смотреть на подклассы и видеть абстрактные базовые классы, а потом перемещать функции из подклассов в базовые классы по той единственной причине, что это удовлетворяет их эстетические потребности. Обычно подобная технология проектирования приводит к ужасным интерфейсам пользователя. Пользователи должны воспринимать модель вашей программы через метафоры. Если элементы интерфейса выглядят, – а главное, ведут себя – как объекты реального мира, то пользователи лучше поймут, как работать с программой, и пользоваться таким приложением легче. Когда вы пытаетесь объединить резко отличающиеся друг от друга объекты реального мира (электронную почту и расписание встреч) в одном элементе интерфейса пользователя, страдает юзабилити, поскольку нет подходящей метафоры из реального мира.

Еще Митчел Капор (Mitchell Kapor) охотно рассказывал всем желающим, что Agenda позволит ввести текст вроде «следующий вторник», а после этого волшебным образом назначить встречу на следующий вторник. Это потрясающе, но любая из приличных программ, написанных за последние десять лет, умеет делать то же самое. Ничего революционного здесь нет.

Команда Chandler также переоценила объем помощи, получаемой от добровольцев. Open source действует не совсем так. Этот метод хорош для реализации «содранных» функций, потому что в этом случае есть спецификация – приложение, которое вы копируете. Он очень хорош для функций, которые не терпится реализовать. Мне нужен аргумент командной строки для EBCDIC, так я добавлю его в код и опубликую. Но если приложение ничего пока не делает, оно никому не интересно. *Никто им не пользуется.* И добровольцев не будет. Почти все разработчики Chandler получают зарплату.

Еще раз приношу глубокие извинения команде Chandler, если Розенберг чего-то не понял или совершенно неверно представил причины задержки, но я и не скрываю, что склонен объяснять такого рода катастрофы ошибками проектирования.

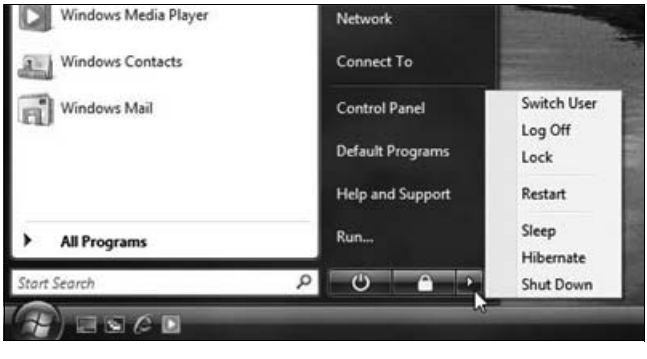
При всем сказанном, такой проект порождает одну хорошую вещь – восхитительную книгу в духе «Души новой машины» и «Шоустоппера» («Showstopper»), рассказывающую о программном проекте, который не удалось довести до конца. Весьма рекомендую.

ГЛАВА ЧЕТЫРНАДЦАТАЯ

Выбор = проблема

21 НОЯБРЯ 2006 ГОДА, ВТОРНИК

Над кнопкой выключения Windows Vista наверняка трудилась целая команда дизайнеров интерфейса, программистов и тестеров, но, положа руку на сердце, разве их решение – лучшее из возможных?



Каждый раз, когда вы хотите завершить работу с компьютером, вам предлагают на выбор девять (пересчитайте сами) вариантов: два значка и семь пунктов меню. Полагаю, два значка служат для упрощенного выбора тех же пунктов меню. Можно догадаться, что значок lock делает то же, что и пункт меню Lock, но я не знаю, какому пункту меню соответствует значок on/off.

На многих ноутбуках есть также четыре комбинации FN+клавиша для выключения питания, перехода в спящий режим, режим пониженного энергопотребления и так далее. Количество вариантов возрастает до тринадцати – кстати, есть еще кнопка on/off (четырнадцать), а еще можно закрыть крышку, итого пятнадцать. Чтобы выключить ноутбук, пользователю приходится выбирать из пятнадцати разных способов.

Чем больше вариантов вы предоставляете, тем труднее пользователю сделать выбор, и тем хуже он себя чувствует. Почитайте, например, книгу Барри Шварца «Парадокс выбора: почему „больше" – это хуже» (Barry Schwartz, «The Paradox of Choice: Why More Is Less», Harper Perennial, 2005). Процитирую рецензию из «Publishers Weekly»: «Шварц, основываясь на собственном обширном опыте социологических исследований, показывает, что ставящее в тупик обилие вариантов захлестывает наш усталый мозг и, в конечном счете, не расширяет наши возможности, а ограничивает их. Мы, американцы, обычно предполагаем, что чем больше вариантов („облегающий" easy fit или „свободный" relaxed fit?), тем счастливее мы будем, но Шварц показывает обратное, доказав, что обилие вариантов выбора вредит нашему психологическому здоровью».

Если каждый раз, чтобы выключить компьютер, приходится выбирать из девяти разных способов (и это только в меню Start, а еще можно нажать кнопку питания на корпусе или закрыть крышку ноутбука), это немного напрягает.

Есть ли выход? Должен быть. В iPod *нет* даже кнопки on/off. Вот некоторые идеи по этому поводу.

Если вам приходилось беседовать с нормальными, а не помешанными на компьютерах людьми, то вы могли заметить, что для них нет разницы между командами Sleep и Hibernate (ждущий и спящий режимы). Эти режимы можно просто объединить. Минус один вариант.

Команды Switch User и Lock (смена пользователя и блокировка) можно объединить, разрешив другому пользователю регистрироваться в системе, когда она заблокирована. Кстати, это должно помешать принудительному завершению сеанса работы пользователя. Еще один вариант долой.

После объединения Switch User и Lock – нужна ли команда Log Off (завершение сеанса)? Все, что дает Log Off, это завершение всех работающих программ. Но то же самое происходит при выключении питания, поэтому, если вам действительно важно завершить все работающие программы, выключите питание, а затем включите его снова. Еще одним вариантом меньше.

Команду Restart (перезагрузка) можно убрать. В 95% случаев перезагрузку предлагает сделать программа установки нового программного обеспечения. В остальных случаях можно просто выключить питание и снова его включить. Еще один вариант отпадает. Чем меньше вариантов, тем легче жить.

Конечно, нужно убрать разделение между значками и меню. Это сократит еще два варианта. В итоге остаются:

Sleep/Hibernate
Switch User/Lock
Shut Down

Что если объединить режимы Sleep, Hibernate, Switch User и Lock? При выборе такого нового режима компьютер показывал бы экран Switch User. Если в течение 30 секунд никто не зарегистрируется, компьютер переходит в ждущий режим. Еще через несколько минут переходит в спячку. В любом случае, он блокируется. Таким образом, остается два варианта:

1. Я отхожу от компьютера.
2. Я отхожу от компьютера и хочу выключить его.

Для чего отключают питание? Если для экономии электроэнергии, то пусть об этом позаботится программа управления электропитанием, она все сделает толковее. Если вы собираетесь залезть внутрь корпуса и боитесь получить удар током, то простого выключения системы для вашей безопасности недостаточно: придется отключить напряжение. Поэтому, если бы Windows использовала энергонезависимую RAM, при простое выгружая оперативную память на флеш-диск, вы фактически могли бы отключать питание в режиме «отсутствия пользователя», не теряя при этом никаких данных. Благодаря новым гибридным жестким дискам это можно было бы делать очень быстро.

В итоге остается единственная кнопка отключения. Назовем ее Пока!. При щелчке по Пока! запирается экран, а содержимое RAM записывается во флеш-память, если это еще не сделано. Вы можете снова вернуться в систему, зарегистрировавшись в ней, и другой пользователь может зарегистрироваться в системе, открыв свой сеанс, а можно и отключить от сети весь компьютер.

Разумеется, вы начнете составлять длинный список убедительных причин, по которым эти многочисленные варианты абсолютно необходимы. Не трудитесь, я и сам их знаю. Без всех этих дополнительных вариантов

никак не обойтись до того момента, когда вам понадобится объяснить своему дядюшке, что для выключения ноутбука нужно выбрать один из пятнадцати вариантов.

Подобный стиль проектирования программ присущ как Microsoft, так и open source, поскольку те и другие стремятся достичь общего согласия, Сделав Счастливыми Всех, но основан он на ложном представлении о том, что людям нравится выбирать из множества вариантов, чего на самом деле мы хотим избежать.

ГЛАВА ПЯТНАДЦАТАЯ

Не только юзабилити

6 сентября 2004 года, понедельник

В течение многих лет самозванные знатоки (вроде меня...) постоянно болтают о юзабилити и о том, что программы должны быть удобными для пользования. Якоб Нильсен знает *математическую формулу*, которую он раскроет вам за 122 доллара, – с ее помощью вы сможете рассчитать показатель юзабилити. (Надо полагать, если полученный показатель юзабилити больше 122 долларов, то вы в выигрыше.)

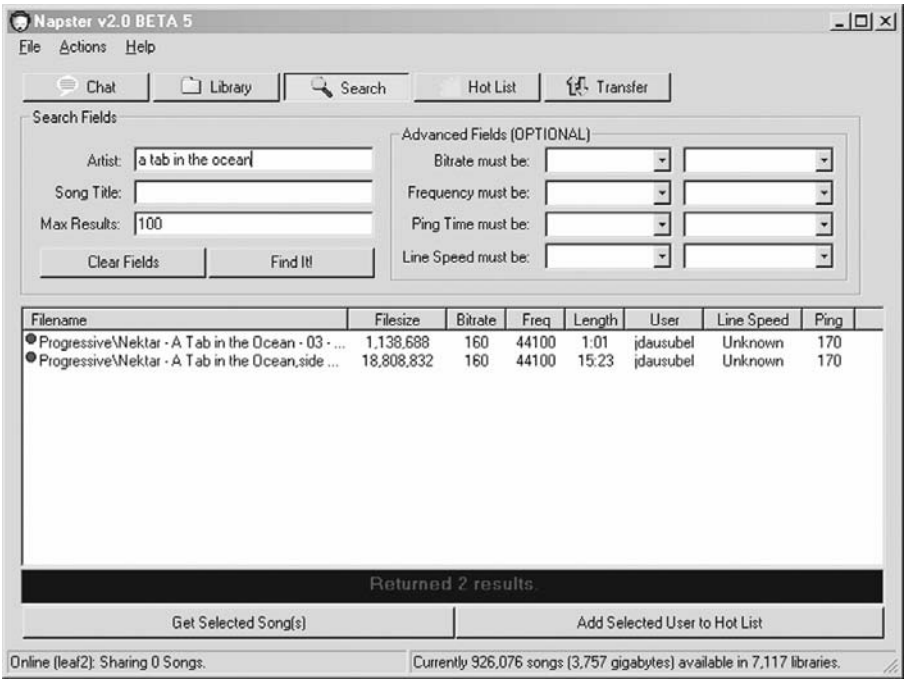
Я написал книгу, которая обойдется вам гораздо дешевле,– «Советы программистам по проектированию пользовательского интерфейса» («User Interface Design for Programmers», Apress, 2001),– в ней описаны некоторые принципы проектирования удобных программ, но формул там нет, и вы покупаете книгу себе в убыток.

На странице 46 этой книги я привел пример, взятый из популярнейшей на тот момент программы Napster. В главном окне Napster с помощью кнопок можно было переключаться между пятью экранами. Согласно принципу юзабилити для *наглядности* (affordance) здесь вместо кнопок следовало применить вкладки (tabs), что я и пытался объяснить.

Несмотря на это, Napster была самой популярной программой на свете.

В рукописи у меня было сказано: «Просто хотелось показать вам, что можно обойтись и без юзабилити», что было довольно странно для книги, посвященной юзабилити. И когда наборщику понадобилось сократить абзац, я с радостью вычеркнул это предложение.

Жуткий момент истины – по крайней мере, для профессиональных разработчиков интерфейсов пользователя – здесь в том, что приложение,



делающее что-то очень важное и нужное людям, будет популярным, даже если с ним крайне неудобно работать. А приложение с идеальным интерфейсом и никому не нужными фунциями обречено на провал. Консультанты по UI спасаются только тем, что разрабатывают все более невероятные формулы ROI для расчета прибыли их клиентов, вложивших 75 000 долларов в проект улучшения юзабилити, – именно потому, что юзабилити воспринимается как нечто «необязательное» (во многих случаях *так оно и есть*). Веб-сайту CNN консультант по юзабилити ничем не поможет. Смею предположить, что в Сети нет ни одного основанного на контенте сайта, которому улучшение его юзабилити поможет увеличить прибыль хотя бы на доллар, потому что сайты, основанные на контенте (я не о тех сайтах, что служат веб-приложениями), сами по себе чертовски практичны.

И все же.

Но сегодня я не собираюсь сетовать на ненужность юзабилити... она *очень* важна в исключительных случаях, и есть масса примеров, когда из-за плохой юзабилити гибли небольшие самолеты, наступал голод и мор и так далее.

Я хочу рассказать о *следующем* уровне задач проектирования программного обеспечения, когда с интерфейсом пользователя все решено: о проектировании *социального интерфейса*.

Видимо, надо пояснить, что я имею в виду.

В 1980-х, когда «изобрели» юзабилити, важной характеристикой программ было качество взаимодействия компьютера с человеком. Для многих программ это по-прежнему важно и сегодня. Но благодаря Интернету появились программы другого типа – те, что обеспечивают взаимодействие между *людьми*.

Форумы, социальные сети, предложение и поиск вакансий (чуть не забыл про электронную почту). Все соответствующие программное обеспечение служит для связи между *людьми*, а не между человеком и компьютером.

Разрабатывая программу для общения между людьми, после того как решите задачи интерфейса пользователя, вы должны решить задачи социального интерфейса. И социальный интерфейс *гораздо важнее*. Даже лучший на свете интерфейс пользователя не спасет программу с неудачным социальным интерфейсом.

Идею социального интерфейса лучше всего иллюстрируют несколько примеров успеха и неудачи.

Несколько примеров

Начнем с примера неудачного социального интерфейса. Каждую неделю я получаю письма от неизвестных с предложением подключиться к их социальной сети. Обычно автор мне незнаком, что слегка раздражает, и я удаляю письмо. Мне объяснили, откуда берутся эти письма: у компаний-производителей программного обеспечения социальных сетей есть специальные инструменты, которые извлекают адреса из вашей адресной книги и отправляют по ним приглашение подключиться к сети. Некоторые почтовые программы сохраняют адреса отправителей всех полученных сообщений, а при подписке на почтовый список рассылки «Joel on Software» вы получаете письмо с просьбой подтвердить свое желание, – и вот уже незнакомцы всех видов запускают программы, которые автоматически просят подтвердить, что я их друг. Спасибо, что подписались на мой бюллетень, но нет, я не познакомлю вас с Биллом Гейтсом. Я твердо придерживаюсь правила не вступать ни в какие из этих социальных сетей,

поскольку меня не устраивает их подход, противоречащий сути реальных сетей человеческого общения.

Теперь взглянем на успешный социальный интерфейс. Многим легче что-то написать, чем сказать в лицо. Подростки меньше смущаются. Им проще назначать свидание, отправив текстовое сообщение на сотовый телефон. Эти программы оказались настолько социально успешными, что сильно улучшили любовную жизнь миллионов (или, по крайней мере, расширили круг их общения). Несмотря на кошмарный интерфейс систем текстовых сообщений, они стали крайне популярны у молодежи. Ирония в том, что *в каждом сотовом телефоне* есть гораздо более удобный интерфейс пользователя для общения между людьми: эта умная вещь называется телефонным звонком. Набираете номер – и другой слышит все, что вы скажете, и наоборот. Так просто. Но в некоторых кругах популярнее неуклюжая система, в которой, ломая пальцы, набираешь длинную строку символов, только чтобы сказать «ты чертовски хороша», потому что эта длинная строка символов приведет к свиданию, а озвучить то же самое тебе никогда не хватит духу.

Другой успешный социальный программный продукт – eBay. Впервые услышав про eBay, я сказал: «Чушь! Из этого ничего не выйдет. Никто не отправит деньги случайному человеку из Интернета, уповая, что тот, по доброте своей душевной, действительно вышлет покупателю какой-то товар». И многие так считали. Мы были трижды неправы. eBay сделал крупную ставку на культурную антропологию человека и *выиграл*. Замечательно в eBay то, что он достиг большого успеха *именно потому*, что в свое время подобная идея казалась ужасной, и никто в нее не верил, а тем временем eBay использовал сетевой эффект и преимущество первопроходца.

Помимо абсолютного успеха или провала у социального программного обеспечения могут быть побочные эффекты. От способа функционирования такой программы в значительной мере зависит тип складывающегося вокруг нее сообщества. У клиентов Usenet есть команда «большое R», позволяющая ответить на сообщение с одновременной вставкой *цитаты из исходного текста*, элегантно помеченной слева знаками >. В старых клиентах телеконференций не было ветвления, поэтому, желая ответить на чье-то конкретное высказывание, вы *вынуждены* были цитировать его с помощью «большого R». Так сложился присущий Usenet стиль дискуссий с точным построчным цитированием, удобный для буквоедов, но мешающий чтению. (Кстати, блогеры-политики, будучи новичками в Интернете и полагая, что открыли нечто новое и интересное, используют ту же техни-

ку, но под новым названием «fisking», о происхождении которого я умолчу. Не бойтесь, все прилично.) Несмотря на то что люди веками вели споры, мелкая особенность программного продукта привела к появлению особого стиля дискуссии.

Небольшие изменения в программе могут существенно повлиять на ее успех или неудачу в достижении социальных целей. Дана Бойд (Danah Boyd) подвергла резкой критике социальные сети Интернета в статье «Autistic Social Software» (*www.danah.org/papers/Supernova2004.html*), обвинив последнее поколение соответствующего программного обеспечения в том, что оно развивает у пользователей склонность к аутизму:

> *Рассмотрим, к примеру, недавний всплеск интереса к социальным сетям, таким как Friendster, Tribe, LinkedIn, Orkut и прочим. Эти технологии пытаются формализировать способы создания и регулирования отношений между людьми. Они позволяют присваивать рейтинг своим друзьям. Иногда они регламентируют способ установления контакта с новыми людьми, предлагая для этого четкую обязательную процедуру.*
>
> *У такого подхода есть свои преимущества – он хорош для программной реализации, но ужас в том, что некоторые принимают его за модель общественной жизни. Он настолько упрощен, что, следуя ему, люди вынуждены вести себя как аутисты, которым для общения необходима определенная процедура. Метод, безусловно, полезен тем, кому показана подобная систематизация, но он далеко не универсален. Что принесет нам технология, предписывающая механистическое установление контактов? Хотим ли мы жить в обществе, где поощряется аутистическое взаимодействие?*

Программное обеспечение, социальный интерфейс которого не учитывает культурную антропологию, на самом деле нелепо, неприятно и не действенно.

Проектирование социального программного обеспечения

Рассмотрим пример проектирования социального интерфейса.

Предположим, ваш пользователь сделал то, чего он делать не должен.

Согласно правилам проектирования юзабилити вы должны сообщить пользователю, в чем его ошибка и как ее исправить. Консультанты по юза-

билити преподносят такой стиль под фирменным названием «защитное проектирование» (Defensive Design).

Это слишком наивный подход для социального программного обеспечения.

Ошибка пользователя может заключаться в том, что он поместил рекламу виагры в каком-нибудь форуме.

Стало быть, вы сообщаете ему «Извините, тема виагры неприемлема. Ваше сообщение не принято».

Что будет дальше? Он все равно опубликует рекламу виагры (или будет долго зудеть по поводу цензуры и Первой поправки).

Проектируя социальный интерфейс, вы должны учитывать социологию и антропологию. В жизни встречаются халявщики, обманщики и прочие негодяи. И среди пользователей социального программного обеспечения найдутся те, кто приспособит его для собственного обогащения за чужой счет. Если не принять мер к явлению, которое экономисты называют *трагедией общин*.

Цель интерфейса пользователя – облегчить жизнь этому самому пользователю, а социальный интерфейс должен облегчить жизнь общества, даже если в результате отдельный пользователь что-то теряет.

Поэтому разработчик хорошего социального интерфейса может решить так: «Не будем показывать никакого сообщения об ошибке. Сделаем вид, что сообщение о виагре принято. Покажем его отправителю, и пусть он, довольный, идет в следующий форум. Но никому другому это сообщение мы показывать не будем».

Действительно, один из лучших способов уйти от атаки – сделать вид, что она успешна. Прикинуться мертвым, в программном смысле.

Нет, это срабатывает не в 100% случаев. Это срабатывает в 95% случаев, сокращая ваши проблемы в 20 раз. Это неопределенная эвристика, как и все прочее в социологии. Но она срабатывает часто, поэтому применять ее стоит, даже если она не абсолютно надежна. Русская мафия поднатужится и придумает схему фишинга, чтобы ее обойти. Простаки с трейлерных стоянок во Флориде в надежде быстро разбогатеть будут действовать по старинке. Авторы 90% спама, который я сегодня получаю, понятия не имеют о фильтрах, так что его не пропустил бы даже жалкий встроенный фильтр Microsoft Outlook, и надо быть полным ламером, чтобы слать спам, который перехватывается хилым поверхностным методом поиска элементарных кодовых фраз.

Маркетинг социальных интерфейсов

Несколько месяцев назад я понял, что создаваемые нами в Fog Creek программы схожи почти навязчивым вниманием к хорошей организации социального интерфейса. Например, в FogBugz есть масса функций (и еще больше *не* функций), цель которых – поощрить *реальный* учет ошибок. Время от времени покупатели сообщают мне, что их прежняя система контроля ошибок простаивала, не соответствуя представлениям людей о совместной работе, но когда появилась FogBugz, люди действительно стали работать с ней, привыкли к ней, и она изменила характер их совместной работы. Сделать вывод о том, что FogBugz действительно используется, можно на основании очень высокого коэффициента обновлений при выходе новой версии, и это значит, что FogBugz не просто пылится на полках, кроме того, даже клиенты, купившие много лицензий, обращаются к нам за новыми, поскольку продукт все шире распространяется в их организации и реально задействован. Я очень горжусь этим. Программы, рассчитанные на коллективное использование, часто не приживаются, так как требуют, чтобы все члены команды одновременно изменили стиль своей работы, что, как вам сообщат антропологи, весьма маловероятно. Учитывая это, в FogBugz внесено много проектных решений, благодаря которым программа оказывается полезной, даже если с ней работает *единственный участник* команды, а также много решений, содействующих постепенному распространению программы среди других участников, пока не будут охвачены все.

Программное обеспечение форума на моем сайте, которое вскоре начнет продаваться как одна из функций FogBugz, создано с еще большей заботой о хорошей поддержке социального интерфейса. В нем есть десятки функций, особенностей и проектных решений, в совокупности обеспечивающих очень высокий уровень интересных дискуссий и лучшее соотношение сигнал/шум среди всех дискуссионных форумов, в которых я когда-либо участвовал. Подробнее я расскажу об этом в следующей главе.

С тех пор я еще больше проникся идеей правильного проектирования социального интерфейса: мы привлекаем таких экспертов, как Клэй Ширки (Clay Shirky), пионер в этой области, вовсю экспериментируем на бедных участниках форума «Joel on Software» (иногда так тонко, что это практически незаметно, например, не показываем сообщение, на которое вы отвечаете, чтобы сократить цитирование и облегчить чтение ветви) и усиленно ищем новые алгоритмы очистки форума от спама.

Новая область

Проектирование социальных интерфейсов все еще в стадии зарождения. Я не видел книг на эту тему, исследованиями в этой области занимается лишь несколько человек и не существует научной дисциплины проектирования социальных интерфейсов. Когда проектирование юзабилити только начиналось, компании-разработчики программного обеспечения привлекали для работы специалистов по эргономике и психологии. Эксперты по эргономике хорошо разбирались в том, какой высоты должен быть рабочий стол, но проектировать графический интерфейс для файловой системы они не умели, поэтому возникла новая область знаний. В итоге проектирование интерфейса пользователя стало самостоятельной дисциплиной, в которой были разработаны такие понятия, как единообразие, наглядность, обратная связь и так далее, ставшие основой научного проектирования интерфейса пользователя.

Полагаю, в ближайшее десятилетие компании для разработки социальных интерфейсов станут нанимать профессиональных антропологов и этнографов. Вместо создания лабораторий юзабилити, они будут на практике вести этнографические наблюдения. Надеюсь, мы определим новые принципы проектирования социальных интерфейсов. Это должно быть увлекательным – как проектирование интерфейса пользователя в 1980-х, – так что следите за новостями.

ГЛАВА ШЕСТНАДЦАТАЯ

Формирование сообществ с помощью программного обеспечения

3 МАРТА 2003 ГОДА, ПОНЕДЕЛЬНИК

В своей книге «Славное местечко» («The Great Good Place», Da Capo Press, 1999) социолог Рэй Ольденбург (Ray Oldenburg) рассуждает о том, что помимо работы и дома человеку нужно еще одно место, где можно встретить друзей, выпить пива, обсудить новости и испытать радость человеческого общения. Кофейни, бары, парикмахерские, пивные, бильярдные, клубы и прочие места времяпрепровождения так же нужны, как фабрики, школы и квартиры. Но капиталистическое общество разрушает такие места, и это обедняет общество. В книге «Боулинг в одиночку» («Bowling Alone», Simon & Schuster, 2001) Роберт Путнэм (Robert Putnam) увлекательно и подробно доказывает, что американское общество практически лишилось третьего места между домом и работой. За последние 25 лет американцы стали «реже вступать в организации, устраивающие сборы своих членов, хуже знать своих соседей, реже встречаться с друзьями и даже общаться со своими семьями». Слишком много стало людей, жизнь которых состоит из работы и сидения дома перед телевизором. Работа–ТВ–сон–работа–ТВ–сон. Мне кажется, что это явление гораздо острее проявляется среди программистов, особенно в таких местах, как Силиконовая долина или окраины Сиэтла. Люди заканчивают колледж, переезжают в другой конец страны, где они никого не знают, и работают по двенадцать часов в день – в основном, из-за одиночества.

Неудивительно поэтому, что очень многие программисты, отчаянно нуждающиеся в человеческих контактах, устремляются в сетевые сообщества: чаты, форумы и проекты open source или играют в Ultima Online. Раз-

рабатывая программное обеспечение для сообществ, мы в какой-то мере пытаемся создать «третье место». И как в любом другом архитектурном проекте, решающее значение имеют конструктивные решения. Если в баре слишком шумно, люди не смогут поговорить. И такое место будет очень сильно отличаться от кофейни. Если в кофейне мало стульев, как в Starbucks, посетители унесут свой кофе туда, где они раньше сидели в одиночестве, вместо того чтобы остаться и пообщаться, как в кофейне сериала «Друзья», который мы смотрим потому, что лучше иметь суррогат места общения, чем никакого вообще.

В программном обеспечении, как и в архитектуре, конструктивные решения влияют на тип сообщества, которое сможет или не сможет образоваться. Люди чаще делают то, что легко сделать, и реже – то, что трудно. Значит, у вас есть способ осторожно поощрять определенные виды поведения, определяющие характер и качество сообщества. Дружеская ли там атмосфера? Ведутся ли серьезные беседы, как в европейском салоне, полном интеллектуалов с интересными идеями? Или это безлюдное место, где по полу разбросаны несколько затоптанных рекламных проспектов, которые никто не удосужился подобрать?

Сравните несколько сетевых сообществ – и вы сразу заметите разницу в общественной атмосфере. Присмотритесь – и вы поймете, что различия часто обусловлены конструктивными особенностями программного обеспечения.

В Usenet какую-то тему могут обсуждать месяц за месяцем, образуя множество новых ветвей, так что трудно сказать, где они начинаются. Как только подключается новичок и задает вопрос по существу, на него обрушиваются старожилы и советуют сначала прочесть FAQ. Цитирование с помощью символа > – сущее наказание при чтении любого потока сообщений, потому что смертельно скучно снова и снова перечитывать всю историю дискуссии, которую вы уже прочли несколько секунд назад. Чтение для маляра Шлемиэля.

В IRC нельзя сохранить за собой ник и канал: как только из комнаты выйдет последний участник, ее может занять кто угодно. Так устроена эта программа. Социальный результат: на следующий день вам часто не удается найти в чате своих вчерашних друзей по комнате, потому что в вашу комнату вас кто-то не пускает, а у друзей другие ники. Единственным способом помешать геененавистникам в Перте (Австралия) перехватывать чат-каналы геев, когда те уходили поспать, оказалось создание программного робота, охранявшего канал 24 часа в сутки. Многие участники IRC

тратят больше сил на сложные войны ботов, попытки захвата каналов и разные дурачества, чем на реальное общение, и часто осложняют жизнь остальным.

На большинстве форумов по инвестированию практически невозможно проследить обсуждение темы с начала и до конца, потому что каждое сообщение отображается на отдельной странице, чтобы показать побольше рекламных баннеров, – задержки при чтении просто сводят с ума. От этого мигающего вокруг диалога коммерческого мусора создается впечатление, будто пытаешься познакомиться на Таймс-сквер, но неоновая реклама совершенно отвлекает внимание.

На Slashdot по каждой теме поступают сотни идентичных ответов, поэтому общение там кажется неинтересным и скучным. Чуть ниже я объясню, почему на Slashdot много одинаковых ответов, а на форуме «Joel on Software» – нет.

На FuckedCompany.com форум совершенно бездарный: подавляющее большинство сообщений содержит неуместную ругань и оскорбления, и все в целом больше напоминает соревнование в грубости, а вовсе не братство.

Итак, мы открыли главную аксиому сетевых сообществ:

Мелкие детали реализации программного обеспечения приводят к крупным различиям в развитии, поведении и атмосфере сообщества.

Пользователи IRC занимаются войной ботов, потому что программа не позволяет резервировать канал. Потоки Usenet в огромной степени избыточны, потому что первая читалка Usenet «rn», разработанная еще в эпоху 300-бодных модемов, не отображает старые сообщения – только новые, поэтому если вы хотите прокомментировать чей-то текст, нужно процитировать его, иначе ваш разбор будет бессмысленным.

С учетом всего сказанного я отвечу на типичные вопросы о форуме «Joel on Software», объяснив, почему он был сделан именно так, как это помогает его работе и что в нем можно усовершенствовать.

В. Почему программа сделана столь упрощенно?

О. В первые дни работы форума «Joel on Software» для начала дискуссии важно было достичь критической массы, избежав эффекта пустого ресторана (никто не заходит в пустой ресторан: все идут в заполненный соседний, даже если он дрянной). Поэтому целью про-

екта было устранение помех для публикации сообщений. Вот почему отсутствует регистрация и нет буквально ни единой функции, поэтому нечего осваивать.

Бизнес-задачей программного обеспечения этого форума было обеспечение технической поддержки продуктов Fog Creek. Это оправдывало его разработку. Для достижения этой цели важнее всего было сделать программу крайне простой, чтобы ей смог пользоваться каждый. Правила форума предельно очевидны. Не встречал никого, кто бы не сразу понял, как пользоваться программой.

В. Нельзя ли сделать такую функцию, чтобы я мог установить флажок «Отправьте мне электронное сообщение, если кто-нибудь прокомментирует мой пост»?

О. Одной этой функцией, которую так легко и потому соблазнительно реализовать, можно погубить любой начинающий форум. Если реализовать ее, можно никогда не достичь критической массы. Такая функция есть у Филипа Гринспена (Philip Greenspun) в LUSENET, и вы можете убедиться, как вянут из-за нее новые дискуссионные форумы.

Почему?

Допустим, человек приходит на форум, чтобы задать вопрос. Если дать ему возможность поставить флажок «Известите меня...», он запостит свой вопрос, установит флажок и больше не вернется. Он просто будет читать ответы, получая их в почтовый ящик. Конец.

Если же убрать такой флажок, человеку не остается ничего иного, как время от времени возвращаться на форум. В какой-то из этих моментов он может увидеть другое сообщение, которое его заинтересует, и у него может возникнуть желание высказаться на эту тему. В критические первые дни, когда нужно активизировать дискуссии, вы усилите «прилипание» к форуму, на нем будет больше людей, и критическая масса будет достигнута значительно быстрее.

В. Хорошо, но нельзя ли хотя бы организовать ветвление? Если кто-то отклоняется от темы, пусть возникнет отдельная ветка, чтобы можно было пойти по ней или вернуться на главную ветвь.

О. Ветвление кажется вполне логичным программисту, но это не тот способ, каким происходит общение в реальном мире. Несвязность ветвящихся дискуссий затрудняет их чтение и отвлекает внимание.

Пример того, как отвлекается внимание: я хочу провести какую-то операцию на веб-сайте моего банка, но сайт работает так медленно, что после перехода на новую страницу я забываю, что хотел сделать. Есть такой анекдот. Беседуют три пожилые дамы. Дама №1: «Я такая забывчивая: как-то на днях стою с сумкой на лестнице около своей двери и не могу вспомнить – то ли пошла вынести мусор, то ли пришла из магазина». Дама № 2: «Я такая забывчивая: сижу как-то в машине возле дома и не могу вспомнить – то ли я возвращаюсь домой, то ли собралась в синагогу». Дама №3: «Слава богу, у меня пока с головой все в порядке, постучу по дереву (тук-тук-тук). Входите, открыто!» Ветвление отвлекает от темы дискуссии, ветвящиеся потоки неестественны и выбивают из колеи. Пусть лучше открывают новую тему, если им нужен офтопик. Вспомнил еще анекдот...

В. Ваш список тем неправильно отсортирован. Он должен сортироваться по времени самого свежего ответа, а не по времени исходного сообщения.

О. Можно сделать и так – это принято на многих веб-форумах. Но тогда некоторые темы будут постоянно вертеться у самого верха, потому что народ готов обсуждать визы H1B или недостатки преподавания вычислительной науки в колледже до бесконечности. Каждый день на форуме появляется сотня новичков, которые читают список сверху и с удовольствием погружаются в эти темы.

У моего способа есть два преимущества. Во-первых, темы быстро опускаются, поэтому общение остается относительно интересным. В конце концов, когда-то нужно прекратить обсуждать одно и то же.

Во-вторых, порядок тем на главной странице сохраняется, так легче найти когда-то заинтересовавшую вас тему, потому что она сохраняет свое положение относительно соседних тем.

В. Почему мне не показывают, какие сообщения я уже читал?

О. У нас лучшая система, которую только можно реализовать распределенным и масштабируемым образом: пусть за этим следит браузер каждого пользователя. Веб-браузеры меняют цвет посещенных ссылок с голубого на фиолетовый. Мы только слегка меняем URL каждой темы, включая в него количество полученных ответов. Благодаря этому при наличии новых ответов сообщение снова выглядит как непрочитанное.

Какие-либо более изощренные способы реализуются труднее и слишком усложняют интерфейс.

В. Эта чертова ссылка «Ответить» находится в самом низу. Пользователю это неудобно, потому что приходится прокручивать всю страницу.

О. Так и задумано. Я хочу, чтобы вы прочли все сообщения, прежде чем отвечать; иначе вы можете написать то, что окажется повторением, или не будет согласовываться с последним постом. Конечно, я не могу физически заставить вас прочесть весь поток, прежде чем принять от вас сообщение, но если поместить ссылку «Ответить» не внизу, а в другом месте, это явно будет способствовать тому, что кто-нибудь разразится своим перлом, не удосужившись прочесть, что было написано до него. Вот поэтому на Slashdot на каждую тему бывает по 500 ответов, из которых интересными окажутся 17, и никто не любит читать дискуссии на Slashdot: это как в классе, когда все дети начинают одновременно выкрикивать один и тот же ответ. («Ха-ха, ... Билл Гейтс! Да это оксюморон».)

В. Эта чертова ссылка «Начать новую тему» находится в самом низу...

О. Ну, ответ тот же.

В. Почему вы не показываете автору сообщение, чтобы он мог подтвердить его перед тем, как оно будет опубликовано? Ведь так можно избежать ошибок и опечаток.

О. Опыт показывает, что это не так. Более того, ситуация просто обратная.

Часть первая: если ввести этап подтверждения, большинство пользователей быстро щелкает мышью, чтобы двигаться дальше. Очень немногие внимательно перечитывают свой текст. Если бы они хотели внимательно перечесть свое сообщение, то могли бы сделать это, пока редактировали его, но им уже не интересен их пост, это прошлогодний снег, и они стремятся вперед.

Часть вторая: отсутствие этапа подтверждения заставляет вести себя осторожнее. Это напоминает исследования, показавшие, что на извилистых горных дорогах лучше убирать защитные заграждения, потому что в их отсутствие водители едут осторожнее, и уж во всяком случае хрупкое алюминиевое заграждение не помешает внедо-

рожнику, движущемуся со скоростью 50 миль в час, сорваться со скалы вниз. Статистика показывает, что сильно напуганный водитель на крутом повороте ползет со скоростью 2 мили в час.

В. Почему я не вижу пост, к которому пишу свой комментарий?

О. Потому что вам бы захотелось процитировать его в своем комментарии. Я делаю все возможное, чтобы сократить объем цитирования и в результате сделать общение более живым, а темы более интересными для чтения. Если цитировать предыдущий текст, тому, кто изучает тему, приходится читать одно и то же несколько раз подряд, что бессмысленно и автоматически наводит тоску.

Иногда люди все же пытаются цитировать, потому что они отвечают на пост, к которому уже есть три комментария, или потому, что они буквоеды, и им хочется возразить на целую дюжину отдельных положений. Нельзя сказать, что это нехорошие люди – просто они программисты, а программирование требует, чтобы над всеми *i* были поставлены точки, – так уж устроен их ум, что не может упустить ни один аргумент, как и проигнорировать ошибку, сгенерированную компилятором. Но будь я проклят, если ОБЛЕГЧУ вам цитирование. Я уже почти нашел способ показывать посты так, чтобы можно было копировать и вставлять их части. Если вам действительно нужно ответить на текст, опубликованный три поста назад, потрудитесь составить приличную фразу типа «Когда Фред сказал то-то, он, вероятно, не учел, что...») и не сорите своими <<<>>>.

В. Почему некоторые посты пропадают?

О. Форум модерируется. Это значит, что несколько человек обладают волшебной силой удаления постов. Если удаляемый пост оказывается первым в потоке, удаляется весь поток, потому что к нему не остается никакого доступа.

В. Но это цензура!

О. Нет, это просто уборка мусора в парке. Если этого не делать, то соотношение сигнал/шум резко упадет. Ведь что пишут – спам и схемы быстрого обогащения, антисемитские комментарии про меня и всякий бред. Идеалистически настроенные юноши могут воображать, что в освобожденном от цензуры мире будет происходить обмен умными мыслями и всеобщий рост IQ, эдакое идеальное Оксфорд-

ское дискуссионное сообщество или Ораторский Уголок. Я прагматик и полагаю, что мир, в котором полностью отсутствует цензура, выглядит как ваш почтовый ящик – 80% спама, рекламы и мошенничества, а немногие интересные люди быстро уйдут.

Если вы ищете место, где можно высказаться без модерирования, советую: (а) создать свой форум и (б) сделать его популярным. (Приношу извинения Ларри Уоллу.)

В. Как вы решаете, что удалять?

О. Прежде всего, я решительно удаляю сообщения не по теме или представляющие, по моему мнению, интерес для слишком малого количества людей. Если что-то, пусть даже очень интересное кому-то, не относится к общим темам, обсуждаемым на «Joel on Software», вряд ли это заинтересует большинство посетителей сайта, которые приходят, чтобы узнать что-нибудь о разработке программ.

Раньше я также включал в «офтопик» обсуждение самого форума, его устройства и юзабилити. Причина здесь несколько иная, близкая к тому, чтобы стать еще одной закономерностью. Любой форум, список рассылки, дискуссионная группа или BBS, независимо от условий, каждую неделю-две принимается обсуждать сам себя. Буквально раз в неделю появляется некто со своим списком усовершенствований программного обеспечения форума и требует внедрить их немедленно. Кто-то отвечает: «Послушай, приятель, ты за это не платишь, Джоэл делает нам одолжение, так что катись». Тогда еще кто-нибудь заявляет: «Джоэл же не по доброте душевной этим занимается, он рекламирует Fog Creek». Это ТАААК СКУЧНО, потому что случается КАЖДУЮ НЕДЕЛЮ. Все равно, что говорить о погоде, когда говорить больше не о чем. Какому-нибудь новичку на форуме это может показаться интересным, но отношение к разработке программного обеспечения имеет слабое, поэтому, как говорит Strong Bad (Злой силач), «DELETED». К сожалению, я убедился, что прекратить обсуждение форума – все равно что остановить течение реки, бесполезно и пытаться. Но все же, если читая эту статью, вы захотите обсудить ее на форуме, очень прошу сделать мне большое одолжение и устоять перед этим соблазном.

Мы также удаляем посты, содержащие личные, ad hominem нападки на непубличные фигуры. Пожалуй, нужно пояснить. «Ad hominem» означает атаку на человека, а не на его идеи. Если сказать «это глупая

идея, потому что...», то все ОК. Но если сказать «ты глуп», то это переход на личность. Злобные, грубые или клеветнические нападки я удаляю. Исключение только одно: поскольку критиковать Джоэла лучше всего на форуме «Joel on Software», злобные или грубые посты, касающиеся Джоэла, не удаляются, но только в тех случаях, когда в них есть хотя бы проблеск полезного аргумента или идеи.

Я автоматически удаляю посты об орфографических или грамматических ошибках автора предыдущего поста. Мы обсуждаем тему интервью, и кто-нибудь норовит прокомментировать: «С таким знанием орфографии у тебя мало шансов найти работу». Крайне скучно обсуждать чьи-то орфографические ошибки. КРАЙНЕ, КРАЙНЕ скучно.

В. Почему бы просто не опубликовать правила, не делая из них загадку?

О. На днях я добирался электричкой из аэропорта Нью-Арк на Манхеттен. Помимо общего обветшалого состояния я обратил внимание на громадный плакат, который очень строго и подробно объяснял, что если вы нарушите правила поведения, вас высадят на следующей станции и вызовут полицию. И я подумал, что 99,99999% тех, кто прочел этот плакат, не собираются нарушать правила, а нарушителям глубоко безразлично, что там написано. В итоге этот плакат только вызовет у честных граждан ощущение, что их в чем-то обвиняют, а социопатов не удержит, постоянно напоминая порядочным гражданам Нью-Джерси, что они находятся в Нью-Арке, центре преступности, где в поездах ездят социопаты и делают нехорошие вещи, за что их ссаживают и вызывают полицию.

Однако почти каждый участник форума «Joel on Software» наверняка и сам догадается, что нехорошо публиковать грубые личные выпады, или посылать в форум для программистов посты об изучении французского языка, или критиковать людей за особенности орфографии. А для остальной 0,01% правила не писаны. Поэтому публикация правил только оскорбляет большинство добропорядочных граждан, ничем не сдерживая дебилов, которые считают, что их дерьмо не пахнет и что их посты не нарушают правила.

Если вы обратитесь к возмутителям спокойствия в открытую, все сочтут вас параноиком или возмутятся тем, что их бранят, хотя они не сделали ничего плохого. Это как снова очутиться в школе, когда один дурак разбил окно, а весь класс должен сидеть и слушать суро-

вую лекцию учителя о том, что бить стекла нехорошо. Поэтому, например, запрещены любые публичные дискуссии по поводу причин удаления какого-либо поста.

В. Почему бы вместо удаления постов не ввести такую систему модерирования, при которой участники голосуют за понравившиеся им посты, и можно выбирать для чтения посты, собравшие больше голосов?

О. На самом деле, так работает Slashdot, но готов поспорить, что 50% тех, кто регулярно читает Slashdot, этого не поняли.

Есть три причины, по которым мне не нравится такая система. Первая: усложняется UI, в котором появляется функция, требующая изучения. Вторая: возникает сложная политика, по сравнению с которой Византийская империя выглядит как школьное самоуправление в третьем классе. И третья: если читать Slashdot, настроив фильтры так, чтобы были видны только интересные посты, ход обсуждения становится совершенно непонятным. Вы получаете кучку случайных разрозненных постов, контекст которых отсутствует.

В. Почему бы не ввести систему регистрации, чтобы отсеивать грубиянов?

О. Как я уже объяснял, форум нацелен на то, чтобы отправлять сообщения было легко. (Напоминаю, что программное обеспечение было создано для технической поддержки.) Регистрация приводит к потере не менее 90% людей, которые могли бы послать сообщение, а в сценарии технической поддержки эти 90% станут звонить по моему бесплатному номеру телефона.

Кроме того, не думаю, что регистрация здесь поможет. Если кто-то ведет себя вызывающе, банить его бесполезно, поскольку можно тривиально зарегистрироваться заново. Идея совершенствования сообщества с помощью регистрации не нова, и она уместна, как мне кажется, в конференциях типа Echo/Well, где не только обсуждается тема, но создается сеть участников и берется плата за членство.

Но требование регистрации НЕ повышает качество ведущихся разговоров или среднее качество участников. Если посмотреть внимательнее на соотношение сигнал/шум на форуме «Joel on Software», можно заметить, что самые шумные авторы (то есть те, кто у кого больше всего слов и меньше всего идей) часто оказываются давними

и стойкими участниками и заходят на форум каждые десять минут. Эти люди испытывают потребность непременно написать «полностью согласен» и высказаться буквально по каждой теме, даже если ничего нового предложить не могут. И такие зарегистрируются обязательно.

В. Каковы планы на будущее?

О. Развитие программного обеспечения форума не является приоритетной задачей для меня и моей компании: оно вполне пригодно и работоспособно и позволило создать интересное место, где можно обсудить сложные проблемы управления в компьютерной области и узнать мысли ряда умнейших людей. У меня слишком много более важных задач для работы. Пусть кто-то другой сделает очередной крупный шаг вперед в обеспечении юзабилити дискуссионных форумов.

Я только что создал форум «New York City», чтобы проверить, способствуют ли территориальные форумы личному знакомству между людьми помимо знакомства в Сети. По моим наблюдениям, региональные сообщества помогут перейти от простого веб-сайта к реальному обществу, настоящему «третьему месту».

Во всяком случае, создание сообщества – прекрасная цель, потому что очень многие из нас очень нуждаются в нем. Будем и дальше трудиться в этом направлении.

ЧАСТЬ ЧЕТВЕРТАЯ

Управление большими проектами

ГЛАВА СЕМНАДЦАТАЯ

Наушники для марсиан

17 МАРТА 2008 ГОДА, ПОНЕДЕЛЬНИК

Сейчас перед вами предстанет главная из всех священных войн, ведущихся в интернет-форумах, где толкутся веб-разработчики. Сталинградская битва в сравнении с ней – все равно что тот случай, когда ваша невестка унеслась с чаепития у бабушки и обернула дерево своим «Мустангом».

Этим предстоящим сражением будет руководить Дин Хашамович (Dean Hachamovitch), ветеран Microsoft, в данное время возглавляющий команду, которая готовит нам очередную версию Internet Explorer – 8.0. Команда IE 8 находится в процессе принятия решения, проходящего в точности по линии разлома между двумя типами мировоззрения. Это различие между консерваторами и либералами, между «идеалистами» и «реалистами», это глобальный джихад между членами одного семейства, программисты против компьютерных теоретиков и «Лексусы» против оливковых деревьев.

А решения нет. Но наблюдать будет очень и очень интересно, потому что 99% участников религиозных войн и не пытаются разобраться в том, что они обсуждают. И это не просто зрелище: это обязательное чтение для разработчиков, проектирующих интероперабельные системы.

Эта война страстей будет вестись вокруг предмета, называемого «стандартами Сети». Пусть Дин сам познакомит вас с существующей проблемой (*blogs.msdn.com/ie/archive/2008/03/03/microsoft-s-interoperability-principles-and- ie8.aspx*):

Во всех броузерах есть режим «Standards» – назовем его «режим Standards», – применяемый для выбора лучшей реализации веб-стандартов, имеющейся в броузере. В каждой версии любого броузера есть собственный режим Standards, поскольку с каждой версией поддержка веб-стандартов улучшается. Есть режимы Safari 3 Standards, Firefox 2 Standards, IE 6 Standards и IE 7 Standards, и все они отличаются друг от друга. Мы хотим, чтобы в IE 8 режим Standards был намного лучше режима IE 7 Standards.

И вся проблема заключается в такой мелочи, как решить, что должен делать IE 8 со страницей, которая объявляет, что поддерживает «стандарты», но на самом деле, вероятно, протестирована только с IE 7.

Да что за штука этот чертов «стандарт»?

Разве нет стандартов во всех инженерных областях? (Есть.)

Разве они обычно не работают? (Ну, это как посмотреть...)

Почему «веб-стандарты» так чудовищно запутаны? (В этом вина не только Microsoft, но и ваша тоже. А также Джона Постела (Jon Postel [1943–1998]). Объясню позднее.)

Решения нет. Любое решение оказывается абсолютно неверным. Эрик Бэнджимен (Eric Bangeman) пишет на Ars Technica: «Команда IE должна пройти по тонкой линии между поддержкой стандартов W3C и обеспечением правильного отображения сайтов, написанных для более ранних версий IE» (*arstechnica.com/news.ars/post/20071219-ie8-goes-on-an-acid2-trip-beta-due-in-first-half-of-2008.html*). Это неверно. Это не тонкая линия. Это линия отрицательной ширины. Тут негде пройти. Они обречены, если пойдут по ней, и обречены, если не пойдут.

Поэтому я не могу и не буду принимать чью-либо сторону в этом вопросе. Но каждый программист-практик должен хотя бы понимать, как действуют стандарты, как они должны действовать и как мы попали в эту заваруху, поэтому я постараюсь немного объяснить, в чем суть проблемы, и вы увидите, что причина та же, по которой так плохо продается Microsoft Vista, и предмет спора тот же, о котором я писал, рассказывая о противостоянии в Microsoft лагеря Рэймонда Чена (Raymond Chen), «прагматиков», и лагеря MSDN, «идеалистов», в котором победил лагерь MSDN, и теперь никто не может понять, куда делись их любимые команды меню в Microsoft Office 2007, и никому не нужна Vista, и идет все тот же спор, в котором вы на стороне идеалистов («красных») или прагматиков («голубых»).

Но начну сначала. Подумаем, *как сделать, чтобы вещи могли работать вместе*.

Какие вещи? Да практически любые. Карандаш и точилка. Телефонный аппарат и телефонная сеть. Страница HTML и веб-броузер. Графическое Windows-приложение и операционная система Windows. Facebook и Facebook-приложение. Стереонаушники и стереосистема.

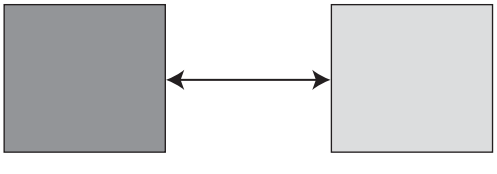
В том месте, где происходит контакт между этими вещами, нужно согласовать много характеристик, иначе они не смогут работать вместе.

Разберу простой пример.

Предположим, что вы отправились на Марс и обнаружили, что у тамошних жителей нет карманных музыкальных плееров. Они все еще пользуются большими музыкальными ящиками.

Вы обнаруживаете огромные возможности для бизнеса и начинаете продавать карманный MP3-плеер (на Марсе его называют Qxyzrhjjjjukltk) и наушники для них. Чтобы подключить к MP3-плееру наушники, вы придумываете аккуратный металлический штекер, вот такой:

Поскольку и плеер, и наушники находятся под вашим контролем, можно гарантировать, что ваш плеер будет работать с вашими наушниками. Это рынок «один-к-одному». Один плеер, одни наушники.



Один к одному

Возможно, вы напишете спецификацию, в расчете что кто-то еще станет изготавливать наушники разных цветов, потому что марсиан очень волнует, какого цвета то, что они вставляют себе в уши.

may not touch the connecting block *f*, an insulating washer *g*, is placed under the screw and washer *h*. The insulating washer is made large enough so that there will be no stray strands to short-circuit the plug. The sleeve *s* is made from brass tubing and passes through the insulating tube *k* to its

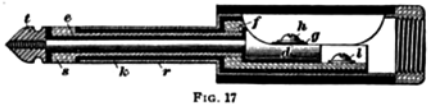
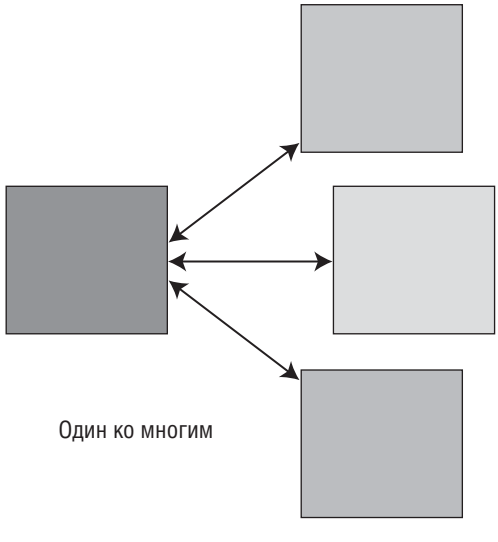


FIG. 17

connecting block *f*, within which it is screwed; the connection is made to the sleeve under the screw and washer *l*. The sleeve connection is made by bending back one of the conductors, when the cord is screwed into the shank of the plug. The strand is thus pinched tightly against the threads,

Но, составляя спецификацию, вы забыли указать, что напряжение должно составлять около 1,4 вольт. Ну, просто забыли. И вот появляется первый честолюбивый изготовитель 100%-совместимых наушников, рассчитанных на 0,014 вольт, а когда он тестирует прототип, взрываются либо его наушники, либо барабанные перепонки слушателя – смотря что прочнее. После некоторого усовершенствования он получает наушники, которые отлично работают – может быть, чуточку хуже, чем ваши.

Появляются все новые и новые изготовители совместимых наушников, и мы оказываемся на рынке «один-ко-многим».



Один ко многим

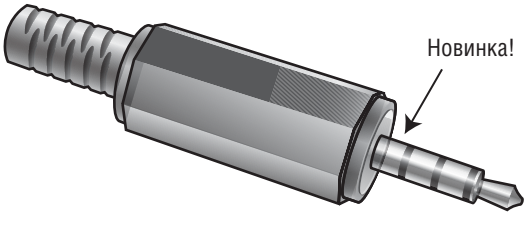
Пока все идет хорошо. Есть стандарт наушников де-факто. Письменная спецификация – неполная и неточная, но каждый желающий производить совместимые наушники может просто воткнуть их в ваш плеер и проверить, работают ли они, и если да, то можно продавать, и все в порядке.

Пока вы не решаете выпустить новую версию, Qxyzrhjjjjukltk 2.0.

В Qxyzrhjjjjukltk 2.0 дополнительно имеется телефон (сотовые телефоны марсианам тоже не удалось изобрести самим), а в наушники встроен микрофон, для чего нужен еще один провод, поэтому вы переделываете разъем в нечто совершенно несовместимое и довольно уродливое, зато с большими возможностями расширения:

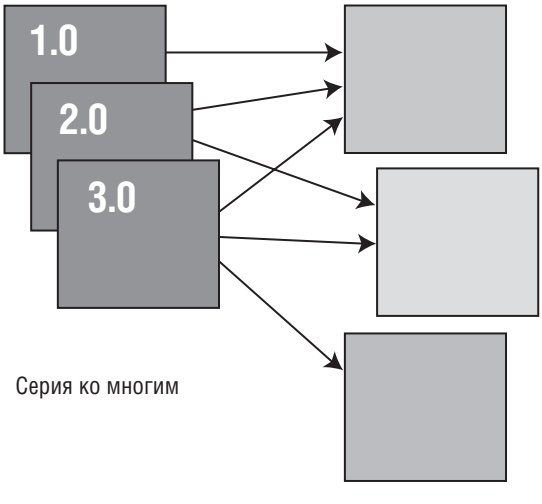
И Qxyzrhjjjjukltk 2.0 терпит полный провал на рынке. Пусть в нем есть замечательный *телефон*, но он не интересует марсиан. Их интересуют огромные коллекции наушников, которые они собрали. Я же не зря говорил, что марсиане очень озабочены цветом того, что вставляют себе в уши. У следящих за модой марсиан накопились целые *шкафы* замечательных наушников. Вам кажется, что все они одинаковые (красные), но марсиане поразительно тонко различают оттенки красного. В рекламе новейших дорогих квартир на Марсе сказано, что в них есть готовые шкафы для наушников. Кроме шуток.

Итак, новый разъем не пользуется успехом, и вы быстро придумываете новую схему:



Обратите внимание, что теперь на центральном стержне имеется дополнительный контакт, обеспечивающий подключение микрофона, но беда в том, что ваш Qxyzrhjjjjukltk 2.1 не знает, какие наушники в него вставляются – с микрофоном или без, а ему нужно это выяснить, чтобы решить, включать ли телефон. Тогда вы изобретаете некий протокол... Новое устройство подает сигнал на контакт микрофона и проверяет, появляется ли он на «земле» – если да, значит это трехконтактный разъем, и микрофона нет, поэтому устанавливается режим обратной совместимости, в котором можно только играть музыку. Вот такой простой протокол согласования.

Теперь это уже не рынок «один-ко-многим». Все стереоустройства производятся одной фирмой, одно за другим, поэтому я назову такой рынок «серия-ко-многим»:



Серия ко многим

Вот несколько знакомых вам рынков «серия-ко-многим»:

1. Facebook: около 20 000 Facebook-приложений
2. Windows: около 1 000 000 Windows-приложений
3. Microsoft Word: около 1 000 000 000 документов Word

Можно привести еще сотни примеров. Главное – помнить, что когда слева появляется новая версия устройства, она должна обеспечивать автоматическую обратную совместимость со всеми старыми аксессуарами справа, рассчитанными на работу со старым устройством, потому что эти старые аксессуары, скорее всего, проектировались без учета появления нового продукта. Марсианские наушники уже изготовлены. Нельзя вернуться назад и все их переделать. Гораздо легче и разумнее переделать новое устройство, чтобы, столкнувшись со старыми наушниками, оно *действовало как старое*.

А поскольку вы хотите двигаться вперед, предлагая новые возможности и функции, то новым устройствам требуется новый протокол, и правильно сделать так, чтобы оба устройства сначала потратили какое-то время на согласование и выяснили, могут ли они оба работать по новейшему протоколу.

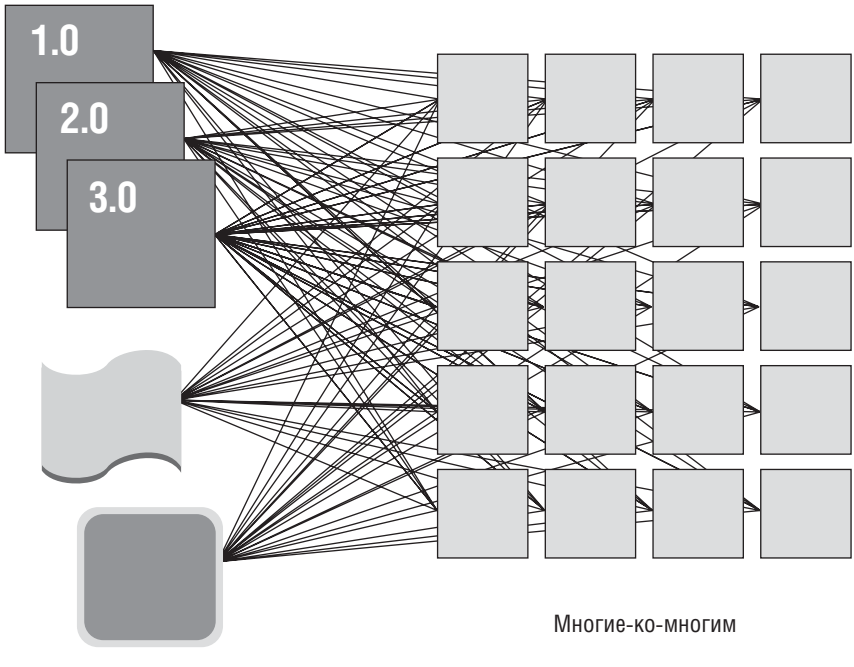
«Серия-ко-многим» – это закон развития мира Microsoft.

Однако есть еще один вариант – рынок «многие-ко-многим».

Проходит несколько лет; Qxyzrhjjjjukltk по-прежнему нарасхват, но на рынке появилось множество клонов Qxyzrhjjjjukltk, типа FireQx с открытым исходным кодом, и множество наушников, и каждый производитель изобретает все новые функции, требующие изменений в конструкции разъема, от чего изготовители наушников сходят *с ума*, потому что им нужно тестировать свои новые изделия с каждым клоном Qxyzrhjjjjukltk, что дорого и долго, и, по правде говоря, у большинства из них нет на это времени, поэтому они проверяют совместимость с самой популярной версией Qxyzrhjjjjukltk 5.0, и если она есть, то удовлетворяются этим; но, конечно, если воткнуть эти наушники в FireQx 3.0, то полюбуйтесь – они взорвутся прямо у вас в руках, потому что никто не может понять до конца одну туманную штуку в спецификации, носящую название *hasLayout*, хотя всем известно, что когда идет *дождь*, свойство *hasLayout* имеет значение «истина», и нужно поднять напряжение, чтобы обеспечить работу щеток-стеклоочистителей, но есть разные мнения по поводу того, считать ли град и снег за дождь по отношению к *hasLayout*, потому что в спецификации о них ничего не сказано. FireQx 3.0 считает, что между снегом или *дождем* нет разницы, потому что стеклоочистители нужны и тогда, когда идет снег, но в Qxyzrhjjjjukltk 5.0 это не так, потому что программист, писавший эту функцию, живет в теплой части Марса, где не бывает снега, да и вообще у него нет водительских прав. Да, на Марсе есть свои водительские права.

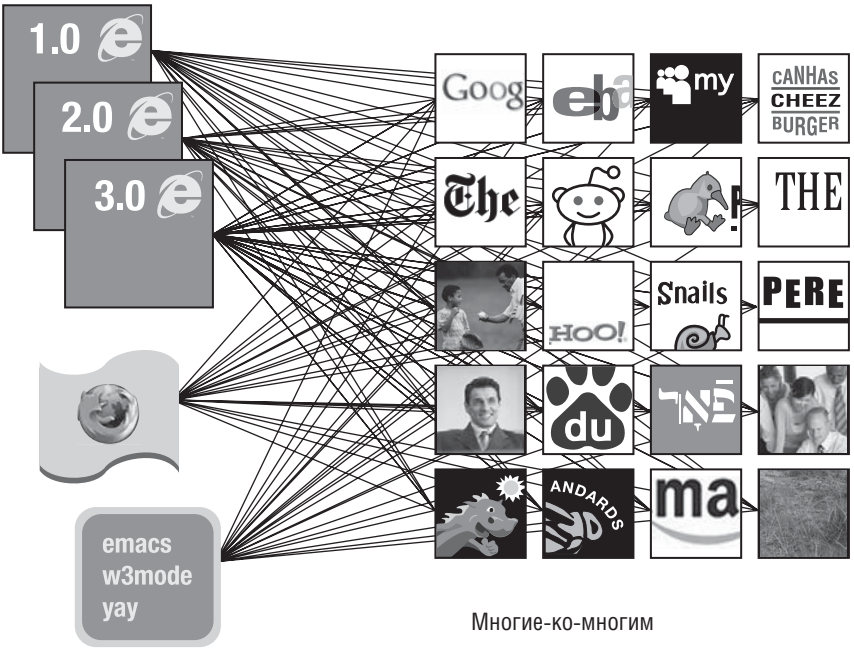
Наконец, какая-то зануда из зануд, воспользовавшись ошибкой, имеющейся в Qxyzrhjjjjukltk 5.0, пространно описывает в своем блоге трюк, благодаря которому Qxyzrhjjjjukltk 5.0 поведет себя так же, как FireQx 3.0, считая, что идет дождь, хотя на самом деле это снег, – для этого надо растопить немного снега, и несмотря на всю нелепость все так и поступят, потому что нужно решить проблему несовместимости *hasLayout*. Но затем разработчики Qxyzrhjjjjukltk в версии 6.0 исправили эту ошибку, то есть вы опять влипли и должны искать еще какую-нибудь ошибку, чтобы с ее помощью обеспечить совместимость своих оснащенных стеклоочистителями наушников с каждым из устройств.

Итак. Вот вам рынок «многие-ко-многим». Слева – множество участников, которые *не* сотрудничают между собой, справа тоже бесчисленное множество участников. И *все они допускают ошибки*, потому что Людям Свойственно Ошибаться.



Многие-ко-многим

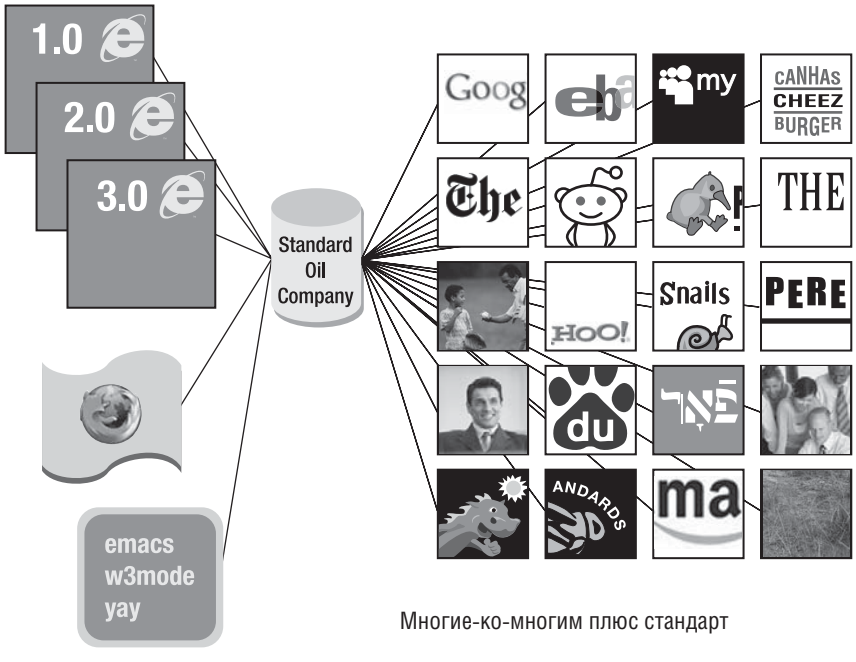
Несомненно, ту же ситуацию мы наблюдаем в HTML. Десятки броузеров и *миллиарды* веб-страниц.



Многие-ко-многим

С течением времени на рынке «многие-ко-многим» все громче раздаются требования выработать «стандарты», чтобы «все участники» (в смысле «мелкие игроки») получили равные возможности показывать правильно все восемь миллиардов веб-страниц, и, что еще важнее, чтобы *разработчики* этих восьми миллиардов страниц могли проверить их всего в одном броузере, применяя «веб-стандарты», и без проверки в *остальных* броузерах убедились, что и те отобразят их страницу правильно.

Как видите, идея в том, что вместо тестирования «многих-ко-многим» появляется тестирование «многие-к-стандарту» и «стандарт-ко-многим», что радикально сокращает количество проверок. Не говоря уже о том, что не нужно добавлять в веб-страницы особый код для обхода ошибок в отдельных броузерах, потому что в этом идеализированном мире не бывает ошибок.



Многие-ко-многим плюс стандарт

Так выглядит идеал.

На практике есть одна проблема, связанная с Сетью: отсутствие возможности протестировать страницу на соответствие стандарту, поскольку не существует эталонной реализации, совместимость с которой гарантировала бы совместимость со всеми остальными броузерами. Нет такой в природе.

Поэтому приходится «тестировать» в голове, проводя чисто мысленный эксперимент, проверяя соответствие ряду стандартизирующих документов, которые вы, вероятно, не читали, а если и читали, то не до конца поняли.

Эти документы *крайне* запутанны. В спецификациях полно выражений вроде «если сестринский структурный блок (который не является плавающим и не позиционирован абсолютно) следует за вставляемым блоком, то последний становится первым строковым блоком структурного блока. Вставляемый блок не может входить в блок, который уже начинается с вставляемого или который сам является вставляемым». Когда я читаю нечто подобное, то начинаю сомневаться, можно ли *вообще* корректно удовлетворить такой спецификации.

Не существует реального способа проверить, удовлетворяет ли спецификации веб-страница, которую вы написали. Есть валидаторы, но они ничего не скажут вам о том, как должна выглядеть страница, а что толку от того, что страница «правильная», если на ней куски текста перекрываются, не выровнены и ничего нельзя разобрать! Поэтому ограничиваются тем, что добиваются правильного вида страницы в одном или двух броузерах. А если есть какая-то ошибка, но в IE и Firefox все выглядит хорошо, то автор даже не заметит, что сделал ошибку.

Но может случиться, что такие страницы будут неправильно показаны в следующих версиях броузера.

Если вам доведется посетить в Иерусалиме какие-нибудь ультраортодоксальные еврейские общины, стремящиеся ни на йоту не отступать от своего Закона, то вы обнаружите, что несмотря на всеобщее согласие относительно того, какая пища является кошерной, раввин одной такой общины не согласится отобедать в доме раввина другой ортодоксальной общины. Веб-дизайнеры открывают для себя то, что было давным-давно известно евреям из Меа Шеарим: общее согласие жить по одной книге не обеспечивает совместимость, потому что законы настолько сложны и запутанны, что почти невозможно разобраться в них в достаточной мере, чтобы избежать ловушек и мин, поэтому для надежности лучше взять блюдо из фруктов.

Стандартизация, конечно, замечательная цель, но чтобы не превратиться в фанатичного приверженца стандартов, следует понять, что из-за человеческого несовершенства стандарты могут неправильно интерпретироваться, оказываясь непонятными и даже двусмысленными.

Суть проблемы в том, что вам кажется, будто существует единый стандарт, но поскольку невозможно проверить соответствие этому стандарту, то он не настоящий: это теоретический идеал вместе с рядом его неверных интерпретаций, а потому такой стандарт не служит решению задачи сократить матрицу тестирования для рынка «многие-ко-многим».

Тег `DOCTYPE` – это миф.

Простой смертный, веб-дизайнер, помещающий на своей веб-странице тег `DOCTYPE`, который должен свидетельствовать о соблюдении им стандарта HTML, впадает в грех гордыни. Он не может знать, соблюден стандарт или нет. Тег надо воспринимать лишь как сообщение о том, что автор *стремился* соблюсти стандарт HTML. *В действительности* он знает только то, что страница показывается в броузерах, которые он проверил – IE,

Firefox, иногда еще Opera и Safari. А может, он просто скопировал тег DOCTYPE из учебника, не понимая, что он означает.

В реальном мире, где живут несовершенные люди, нельзя установить стандарт с помощью одной лишь спецификации: *должна* существовать еще очень строгая эталонная реализация, которую все должны применять для тестирования. В противном случае у вас получится дюжина разных «стандартов», а это все равно что ни одного.

Джон Постел спровоцировал проблему в 1981 году, сформулировав принцип надежности: «Будьте консервативны в том, что делаете сами, и либеральны в том, что принимаете от других» (*tools.ietf.org/html/rfc793*). Этим он хотел сказать, что для надежной работы протоколов нужно максимально точно придерживаться спецификаций самому и в то же время быть снисходительнее к партнерам, не вполне следующим спецификации, если вы в состоянии понять, что они хотят сказать на самом деле.

Так, для того чтобы сделать абзац с маленьким текстом, формально нужно написать <p><small>, но многие пишут <small><p> что формально неверно, хотя большая часть веб-разработчиков этого не понимает, но веб-броузеры «прощали» им это, делая текст маленьким, понимая, что именно это имелось в виду.

Так и появились все эти страницы с ошибками, потому что все разработчики первых броузеров сделали суперлиберальные, дружелюбные и терпимые программы, которые любили вас за то, *какие вы есть*, и не наказывали за ошибки. И мы получили кучу ошибок, и принцип Постела оказался неэффективен. На эту проблему не обращали внимания долгие годы. В 2001 году Маршалл Роуз (Marshall Rose) наконец-то написал (*tools.ietf.org/html/rfc3117*):

> *Вопреки интуиции, принцип надежности Постела (будь консервативен в том, что отправляешь, и снисходителен к тому, что принимаешь) часто ведет к проблемам в развертывании решений. Почему? Когда выходит новая реализация чего-либо, она сначала сталкивается лишь с ограниченным набором существующих реализаций. Если эти реализации следуют «принципу надежности», то ошибки в новой реализации могут остаться незамеченными. Тогда новая реализация получает некоторое, хотя и не очень широкое, распространение. С появлением новых реализаций процесс неоднократно повторяется. В конце концов не вполне корректная реализация наталкивается на другие реализации, настроенные не*

столь либерально, как предшествующие. Легко представить себе, что произойдет.

Нужно отдать должное Джону Постелу за его огромный вклад в появление Интернета и не стоит винить его за этот несчастный принцип надежности. 1981 год – давняя история. Если бы Постел знал, что 90 миллионов людей без специальной подготовки и отнюдь не программистов станут создавать веб-сайты и делать это со всевозможными ошибками, а первые броузеры ввиду ошибочного милосердия их авторов станут прощать эти ошибки и все равно отображать страницу, он бы понял, что его принцип не верен и что на самом деле идеалисты веб-стандартов правы, и Сеть «должна была» строиться на очень и очень строгих стандартах, и каждый броузер должен был быть решительно *несносным*, показывая *все ошибки*, а разработчика, не сумевшего «быть консервативным в том, что делаешь», следовало лишать возможности публиковать свои страницы *где бы то ни было*, пока не научится правильно работать.

Но, конечно, если бы это произошло, возможно, у Сети могло и не быть такого взлета, а мы все, скажем, пользовались бы гигантской сетью Lotus Notes под управлением AT&T. (Бр-р-р.)

Если бы да кабы... Что теперь говорить. Мы там, где мы есть. Мы не можем изменить прошлое – только будущее. Да и будущее-то изменить не так просто.

Если бы вы были прагматиком в команде Internet Explorer 8.0, то в вашей памяти были бы выгравированы слова Реймонда Чена. Он описывал, как Windows XP пришлось эмулировать ошибочное поведение старых версий Windows (*blogs.msdn.com/oldnewthing/archive/2003/12/23/45481.aspx*):

Поставьте себя на место покупателя. Вы купили программы X, Y и Z. Потом вы обновились до Windows XP. В результате ваш компьютер стал регулярно зависать, а программа Z не работает вообще. Вы скажете друзьям: «Не обновляйтесь до Windows XP. Она виснет и несовместима с программой Z». Вы же не будете заниматься отладкой, чтобы выяснить, что зависание вызывает программа X, а программа Z не работает потому, что использует недокументированные сообщения окон! Вместо этого вы просто вернете Windows XP и потребуете назад деньги. (Вы купили программы X, Y и Z достаточно давно, и 30-дневный период для возврата истек. Так что все, что вы можете вернуть, – это Windows XP.)

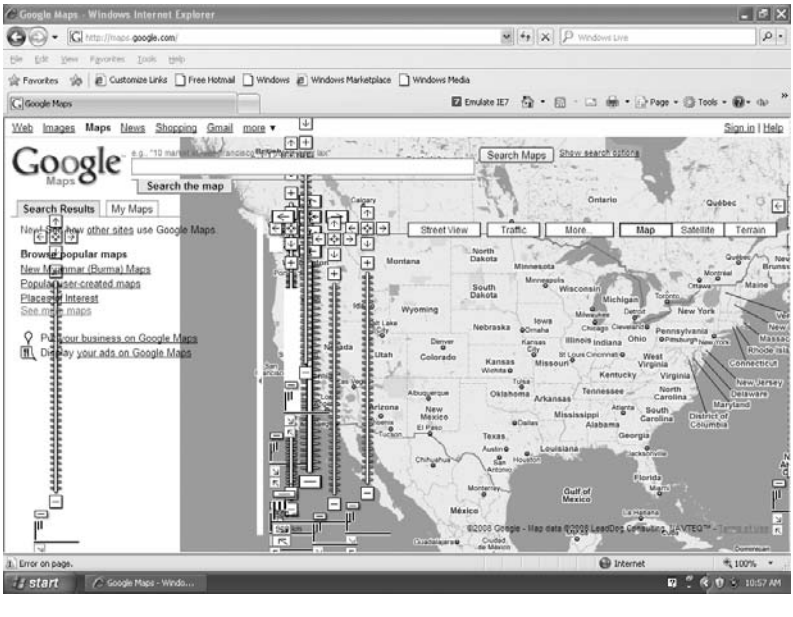
Сразу приходит в голову... хм-м-м... что сегодня можно написать иначе:

Поставьте себя на место покупателя. Вы купили программы X, Y и Z. Потом вы обновились до Windows ~~XP~~Vista. В результате ваш компьютер стал регулярно зависать, а программа Z не работает вообще. Вы скажете друзьям: «Не обновляйтесь до Windows ~~XP~~Vista. Она виснет и несовместима с программой Z». Вы же не будете заниматься отладкой, чтобы выяснить, что зависание вызывает программа X, а программа Z не работает потому, что использует ~~недокументированные~~небезопасные сообщения окон! Вместо этого вы просто вернете Windows ~~XP~~Vista и потребуете назад деньги. (Вы купили программы X, Y и Z достаточно давно, и 30-дневный период для возврата истек. Так что все, что вы можете вернуть, – это Windows ~~XP~~Vista.)

Победа идеалистов над прагматиками в Microsoft, о которой я говорил в 2004 году, – прямая причина того, что Vista получает ужасные отзывы и плохо продается.

А как это относится к команде IE?

Поставьте себя на место покупателя. Вы посещаете 100 сайтов в день. Затем вы обновились до IE 8. И теперь половина страниц неправильно отображается, а Google Maps не работает вообще.



Вы скажете друзьям: «Не обновляйтесь до IE 8. Он неправильно отображает страницы, а Google Maps не работает вообще». Вы же не станете просматривать исходный код страницы, чтобы определить, что на сайте *X* – нестандартный HTML, а Google Maps не работает потому, что использует нестандартные объекты JavaScript из старых версий IE, которые так никогда и не были одобрены комиссией по стандартизации? Вместо этого вы просто удалите IE 8. (Сайты вам неподвластны. Те, кто разрабатывал некоторые из них, уже умерли. Так что все, что вы можете сделать, – это вернуться к IE 7).

Итак, если вы разработчик в команде IE 8, то первым вашим побуждением будет сделать то, что всегда работало на рынках «серия-ко-многим». Вы решите добавить в протокол небольшое согласование, чтобы продолжать эмулировать старое поведение для каждой страницы, которая не сообщит *явно*, что она ожидает нового поведения, так что все существующие страницы будут работать по-прежнему, а новое красивое поведение достанется только тем сайтам, которые поставили у себя флажок, говорящий программе: «Эй! Я понимаю IE 8! Пожалуйста, осчастливьте меня Благодатью IE 8!»

И действительно, именно о таком решении команда IE сначала объявила 21 января. Веб-броузер должен был молча поддерживать существующие страницы, действуя как старый, напичканный ошибками и ненавидимый веб-разработчиками IE 7, чтобы никому не пришлось модифицировать свой сайт.

Прагматичный программист сказал бы, что первое решение команды IE было правильным. Но молодые идеалисты «стандартов» взбунтовались.

Они заявили, что IE должен предоставлять поведение по стандартам *без* особого тега «Эй! Меня протестировали в IE 8». Их тошнило от специальных тегов. На каждой чертовой странице нужно было сделать тридцать семь уродливых хаков, чтобы ее было нормально видно в пяти или шести популярных броузерах. Хватит уродливых хаков! Будь они прокляты, эти восемь миллиардов страниц!

И команда IE сделала резкий поворот. Их вторым – надо думать, не последним – решением стал идеалистический подход: обращаться со всеми сайтами, заявляющими о своей «совместимости со стандартами», как с разработанными и протестированными для IE 8.

Почти все сайты, которые я посетил в IE8, отображались с теми или иными искажениями. Сайты, на которых много JavaScript, обычно вообще мертвы. У некоторых страниц некоторые проблемы отображения: что-то

не на том месте, вместо всплывающего меню – раскрывающееся, а посредине – откуда-то взявшиеся полосы прокрутки. На некоторых сайтах проблемы видны не сразу: они выглядят нормально, но потом оказывается, что какая-нибудь важная форма не отсылается или ведет на пустую страницу.

И это ошибки *не страниц.* Обычно это веб-сайты, тщательно построенные в соответствии со стандартами Сети. Но ни IE 6, ни IE 7 в действительности не соблюдали эти стандарты, поэтому данные сайты содержат маленькие хаки, типа «в Internet Explorer: подвинуть этот блок на 17 пикселов вправо, чтобы компенсировать ошибку в IE».

IE 8 – это тоже IE, но у него больше нет бага, имевшегося в IE 7 и двигавшего ту штуку на 17 пикселов влево от того места, где она должна была быть по стандарту. И теперь код, который был вполне оправданно написан, не работает.

IE 8 не в состоянии правильно отобразить большинство страниц, пока вы не сдадитесь и не нажмете кнопку «ЭМУЛИРОВАТЬ IE 7». Но идеалистам все равно: они считают, что все эти страницы нужно переписать.

Некоторые страницы невозможно переписать. Одни могут находиться на CD-ROM. Другие написаны теми, кого нет в живых. Большинство из них создано людьми, не имеющими ни малейшего понятия, что происходит и почему их страница, за которую они заплатили разработчикам 4 года назад, перестала работать.

Идеалисты возрадовались. Сотни их снизошли до блога IE, чтобы впервые в жизни написать что-то хорошее о Microsoft.

Я посмотрел на часы...

Тик-тик-тик.

Делом секунд было подождать пока в форумах начнут появляться сообщения вроде следующего (*forums.microsoft.com/MSDN/ShowPost.aspx?PostID=2972194&SiteID=1*):

> *Я скачал IE 8 и баги вместе с ним. Некоторые из моих сайтов, например «HP», трудно читать, потому что вся страница очень маленькая... Скорость соединения с Интернетом тоже иногда падала. При работе с Google Maps полно наложений, поэтому очень неудобно!*

Да-а-а... Все самодовольные идеалисты смеются над этим новичком-идиотом. Но потребитель не идиот. Это ваша жена, к примеру. Так что перестаньте смеяться. 98% пользователей в мире поставит IE 8 и скажет: «В нем

ошибки, и я не вижу своих страниц». И они абсолютно не разделяют глупый религиозный энтузиазм по поводу выпуска веб-броузера, который строго следует мистическому идеалистическому «стандарту», нигде реально не воплощенному. Им не интересны ваши объяснения по поводу хаков, портящих страницы. Им нужен броузер, который работает с реальными сайтами.

Итак, вы видите ужасающий пример пропасти между двумя лагерями.

Лагерь веб-стандартов напоминает троцкистов. Вы думаете что они – левое крыло, но если вы сделали веб-сайт, который претендует на поддержку стандартов, но на деле оказывается не так, то идеалисты превращаются в Джо Арпайо (Joe Arpaio), самого жестокого шерифа Америки. «ТЫ СДЕЛАЛ ОШИБКУ И ТВОЙ САЙТ ДОЛЖЕН СЛОМАТЬСЯ! Мне плевать, что 80% ваших сайтов перестанет работать. Я вас всех засажу в тюрьму, и вы будете носить розовые пижамы, есть сэндвичи за 15 центов и работать сидя на цепи. И мне плевать, если вся страна окажется в тюрьме. Закон есть закон!»

С другой стороны – прагматичный, чувствительный, теплый, белый и пушистый программист. «Почему бы не сделать режим IE 7 режимом по умолчанию? Одна строчка кода... Вжик! Готово!»

Раскрыть вам тайну? Я думаю, все будет так. Команда IE 8 скажет всем, что в IE 8 по умолчанию будет установлен режим веб-стандартов, и начнет долгое бета-тестирование, на протяжении которого будет просить всех проверить свои страницы в IE 8 и постараться заставить их работать. А ближе к выпуску окончательной версии, когда окажется, что только 32% всех страниц в мире отображается корректно, они скажут: «Ребята, нам очень жаль, мы действительно хотели, чтобы режим стандартов был установлен по умолчанию, но мы не можем поставлять неработающий броузер» и вернутся к прагматичному решению. Или не вернутся, потому что прагматики в Microsoft уже давно не у дел. В этом случае IE потеряет значительную часть рынка, что бесконечно обрадует идеалистов, а годовая премия Дина Хашамовича, скорее всего, не уменьшится ни на цент.

Видите? Правильного ответа нет.

Как обычно, идеалисты на 100% правы в принципах и, как обычно, прагматики правы на практике. Флейм будет продолжаться годами. Этот спор делит мир ровно пополам. Если вы знаете, где купить акции Священных Войн Интернета, то сейчас самое время это сделать.

ГЛАВА ВОСЕМНАДЦАТАЯ

Почему форматы файлов Microsoft Office такие сложные (и как это обойти)

19 ФЕВРАЛЯ 2008 ГОДА, ВТОРНИК

На прошлой неделе Microsoft опубликовала форматы двоичных файлов Office. Выглядят они совершенно безумно. Описание форматов Excel 97–2003 – это PDF-документ из 349 страниц. Погодите, это еще не все! В документе есть такой интересный комментарий:

Каждая книга Excel хранится в составном файле.

Итак, файлы Excel 97–2003 – это составные (compound) OLE-документы , то есть, по сути, представляют собой файловую систему, заключенную в один файл. Это достаточно сложная система, и чтобы разобраться с ней, вам придется дополнительно прочесть 9-страничную спецификацию. Все эти «спецификации» больше похожи на структуры данных языка C, чем на спецификацию в привычном нам смысле. Это целая иерархическая файловая система.

Если вы взялись за эту документацию в надежде за выходные написать какой-нибудь хитрый код для импорта документов Word в свой блог или чтобы вывести данные личного финансового учета в виде таблицы Excel, сложность и объем спецификации быстро отобьют у вас это желание. Нормальный программист придет к выводу, что двоичные форматы Office:

- умышленно запутаны,
- выдуманы безумным киборгом,
- созданы крайне неумелыми программистами,
- корректно прочесть или записать их невозможно.

Во всех четырех случаях вы бы ошиблись. Проведем небольшое исследование, и я покажу вам, почему эти форматы так усложнились, почему они не свидетельствуют о непрофессионализме Microsoft и как все это обойти.

Форматы рассчитаны на очень старые компьютеры. В первых версиях Excel для Windows один мегабайт ОЗУ считался приемлемым объемом памяти, а двадцати мегагерц процессора 80386 было достаточно для комфортной работы Excel. Форматы файлов Microsoft существенно оптимизированы с целью ускорить открытие и сохранение файлов:

- Это двоичные форматы, так что загрузка записи обычно сводится к копированию (blitting) некоторого сегмента байтов с диска в память, где они заполняют те самые структуры C, с которыми можно работать. Никакого лексического и синтаксического разбора, которые выполняются на порядок медленнее, чем копирование.
- Формат задуман так, чтобы стандартные операции выполнялись быстро. Например, в Excel 95 и 97 есть режим Simple Save, применяемый как более быстрый вариант составного OLE-документа, если обычный документ оказывается недостаточно быстрым. В Word есть такая функция, как Fast Save: чтобы быстро сохранить длинный документ, сделанные изменения чаще всего дописываются в конец файла, чтобы не переписывать весь файл заново. На старых жестких дисках это означало, что большой документ записывался в течение одной секунды вместо тридцати. (Правда, при этом в файле сохранялись удаленные из документа данные. Не всем это понравилось.)

Форматы были рассчитаны на использование библиотек. Если вам нужно было написать собственную программу импорта, требовалось обеспечить поддержку таких вещей, как Windows Metafile Format (для графики) и составные OLE-документы. Если вы работаете в Windows, то все тривиально, поскольку соответствующие библиотеки там есть. Воспользовавшись ими, команда Microsoft облегчила себе задачу. Но если вы пишете что-то свое с чистого листа, вам придется реализовать все это самостоятельно.

Office хорошо поддерживает составные документы: например, в документ Word можно вставить таблицу Excel. В идеале программа импорта файла Word должна разумно обработать встроенную таблицу Excel.

Обмен файлами с другими приложениями не предполагался. В то время считалось, что файл в формате Word следует читать и записывать только с помощью программы Word. Из этого следовало, что разработчи-

кам Word, если они хотели изменить формат, требовалось позаботиться лишь о том, чтобы: а) формат работал быстро; б) *в базовый код Word* нужно было внести минимум изменений. Такие идеи, как SGML и HTML – стандартизированные, пригодные для обмена данными форматы, – возникли лишь с появлением возможности обмена документами через Интернет; двоичные форматы Office разрабатывались за десятилетие до этого. Всегда предполагалось, что для обмена документами есть экспорт и импорт. На самом деле, в Word имеется формат RTF, разработанный для облегчения обмена документами, и он там есть практически с самого начала. RTF полностью поддерживается и сейчас.

Форматы должны отражать все сложные особенности приложений. Каждый флажок в окнах диалога, каждое свойство форматирования, каждая функция Microsoft Office должны быть записаны где-то в файле. В окне Абзац есть флажок не отрывать от следующего, который задает перенос абзаца на новую страницу, если требуется, чтобы он был на одной странице со следующим абзацем. Состояние этого флажка должно быть учтено в формате. А это значит, что если вам нужно написать точный клон Word, способный корректно читать документы Word, вы должны реализовать и эту функцию. Создавая конкурентоспособный текстовый процессор, который должен загружать документы Word, вы за минуту напишете код, считывающий этот бит из файла. Но чтобы учесть этот флажок, вам придется переделать свой алгоритм создания макета страницы, на что могут уйти недели. Если вы не сделаете этого, то пользователи вашего клона, открыв в нем свои файлы Word, увидят искаженные страницы.

Форматы должны отражать историю развития приложений. Многие особенности формата связаны со старыми, сложными, противными и редко используемыми функциями. Они сохранены ради обратной совместимости, а еще потому, что Microsoft ничего не стоит оставить старый код. Но если вы действительно хотите тщательно обрабатывать эти файлы, вам придется заново повторить всю ту работу, которую 15 лет назад сделали в Microsoft какие-нибудь практиканты. Вывод из этого такой: в нынешние версии Word и Excel вложены уже тысячи человеко-лет труда, и если вы хотите их точно клонировать, вам придется работать тысячи лет. Формат файла – это лишь краткая сводка функций, поддерживаемых приложением.

Рассмотрим подробно один маленький пример. Файл Excel – это набор записей формата BIFF разных типов. Посмотрим, что сказано в спецификации про первую запись BIFF. Она называется 1904.

Спецификация Excel по поводу этой записи крайне немногословна: запись 1904 указывает на «использование системы дат 1904». Классический пример бесполезной спецификации. Когда программист, работающий с форматом Excel, обнаруживает в спецификации формата такой текст, он может заподозрить, что Microsoft здесь что-то скрывает. В этой информации недостаточно информации. Нужно знать еще что-то, о чем я вам сейчас и поведаю. Есть два вида таблиц Excel: с датами, отсчитываемыми от 1 января 1900 года (вместе с умышленной ошибкой, которая считает 1900 год високосным, сделанной для совместимости с Lotus 1-2-3, о чем здесь неинтересно рассказывать), и с датами, отсчитываемыми от 1 января 1904 года. Excel поддерживает обе системы, потому что первая версия писалась для Маков, где операционная система использовала отсчет дат от 1904 года, и так было проще, а Excel для Windows должна была импортировать файлы Lotus 1-2-3, где отсчет велся от 1/1/1900. Тут есть от чего заплакать. Когда только программисты не ошибались, и вот вам пример.

Вы можете встретить файлы обоих типов – и 1900, и 1904, обычно в зависимости от того, где они были созданы – в Windows или на Маке. Преобразование из одного формата в другой может нарушить целостность данных, поэтому Excel не станет менять тип файла автоматически. Если вы хотите работать с файлами Excel, вам придется реализовать поддержку обоих типов. И тут загрузкой одного бита из файла не обойтись. Вам придется переписать весь код отображения дат и разбора, чтобы поддерживать оба формата отсчета дат. Думаю, это займет несколько дней.

Далее, если вы пишете клон Excel, то перед вами возникнет масса мелких и скрытых проблем обработки даты. Когда Excel конвертирует числа в даты? Как работает форматирование? Почему «1/31» интерпретируется как 31 января текущего года, а «1/50» – как 1 января 1950 года? Если описывать все эти тонкости поведения, придется написать документ, по объему информации сравнимый с исходными текстами Excel.

И это только первая из сотен записей BIFF, которые вам придется поддерживать, и одна из простейших. Многие из них настолько сложны, что доведут до слез и опытного программиста.

Из этого только один вывод. Очень мило, что Microsoft опубликовала формат файлов Office, но задача импорта или сохранения в форматах Office от этого не стала проще. Программы Office безумно сложны и богаты функциями, и нельзя рассчитывать, что, реализовав 20% наиболее попу-

лярных, вы сможете осчастливить 80% пользователей. Спецификация двоичных файлов, по сути, сохранит только несколько минут, потраченных на дизассемблирование чрезвычайно сложной системы.

Да, я обещал рассказать об обходных путях. Прежде всего, в большинстве популярных программ попытки читать и записывать данные в двоичных форматах Office – результат ошибочного решения. Есть две основные альтернативы, заслуживающие внимания: возложить эту обязанность на Office либо воспользоваться более простыми форматами файлов.

Заставить Office выполнять тяжелую работу. Word и Excel основаны на очень сложных объектных моделях, доступных посредством COM-автоматизации, что позволяет программно делать *все*. Во многих случаях проще использовать код Office, чем пытаться заново реализовать его. Вот несколько примеров.

1. У вас есть веб-приложение, которое должно выводить ваши файлы Word в формате PDF. Я бы это реализовал так: несколько строк кода VBA для Word загружают файл и сохраняют его путем экспорта в PDF, имеющегося в Word 2007. Этот код можно запустить напрямую, например из ASP или ASP.NET под IIS. И это будет работать. Первый раз Word будет запускаться несколько секунд. Дальше будет быстрее, потому что подсистема COM держит Word в памяти несколько минут на случай, если он потребуется снова. Все это достаточно быстро для веб-приложения.
2. То же самое, но хостинг на Linux. Купите один сервер под Windows 2003, установите на него лицензионный Word и создайте маленький веб-сервис, который сделает все, что нужно. Полдня работы на C# и ASP.NET.
3. То же самое, но требуется масштабировать решение. Возьмите столько машин, описанных на шаге 2, сколько нужно, и поставьте балансировщик нагрузки. Программирование не требуется.

Такой подход работает для большинства офисных задач, которые могут выполняться на вашем сервере. Например:

- Открыть книгу Excel, записать данные во входные ячейки, произвести пересчет и получить результаты в выходных ячейках.
- Использовать Excel для генерации диаграмм в формате GIF.
- Извлечь любую информацию из любой таблицы Excel, не ломая себе голову над форматами файлов.

- Конвертировать файл Excel в формат CSV (другой способ – использовать ODBC-драйверы для Excel и получить данные через SQL-запрос).
- Редактировать документы Word.
- Заполнять формы Word.
- Конвертировать данные между многочисленными форматами, поддерживаемыми Office (имеются средства импорта для десятков форматов текстовых процессоров и электронных таблиц).

Во всех этих случаях есть способы сообщить объектам Office, что они работают не в интерактивном режиме и не должны перерисовывать экран, как и требовать ввода данных пользователем. Кстати, если вы хотите идти этим путем, есть несколько ловушек и нет официальной поддержки, так что сначала прочтите статью в базе знаний Microsoft (*http://support.microsoft.com/default.aspx?scid=257757*).

Воспользоваться более простым форматом для записи файлов. Если вам нужно только программно *создавать* документы для Office, есть много других форматов, которые Office уверенно откроет, не пропустив ни байта.

- Если вам нужно записать данные в таблицу для использования в Excel, попробуйте CSV.
- Если нужны табличные расчеты, которые CSV не поддерживает, то есть формат формат WK1 (Lotus 1-2-3), который намного проще и отлично открывается в Excel.
- Если совершенно необходимо создавать именно файлы Excel, найдите какую-нибудь древнюю версию, например 3.0, в которой нет всех этих составных документов, и сохраните минимальный файл, в котором есть только нужные вам функции. По этому файлу определите, какой минимум BIFF-записей вам потребуется выводить, и изучите только соответствующую часть спецификации.
- Если нужно вывести документы для Word, можно использовать HTML. Word хорошо открывает файлы в этом формате.
- Если требуется сложное форматирование документа для Word, ваш выбор – RTF. Все, что есть в Word, можно записать в RTF, но это текстовый, а не двоичный формат, поэтому можно изменить данные в RTF-документе, сохранив корректность файла. Например, создаете в Word красиво отформатированный документ с фиктивными данными в нужных полях и с помощью простой текстовой замены ди-

намически заменяете их. Этот RTF-документ будет отлично открываться в любой версии Word.

Во всяком случае – если только вы действительно не собрались выпустить программу-конкурент Office, которая будет идеально считывать и записывать все документы Office, и тогда у вас впереди тысячи человеко-лет работы, – чтение и запись двоичных форматов Office наверняка окажется самым трудоемким этапом в решении вашей задачи.

ГЛАВА ДЕВЯТНАДЦАТАЯ

Где грязь, там и деньги

6 декабря 2007 года, четверг

Когда в юности я работал в хлебопекарне, проклятием для меня было тесто. Оно было липкое, приставало всюду, и его было трудно отчистить. Придя домой, я находил комочки теста у себя в волосах. Каждую смену часа два уходило на то, чтобы очистить от теста механизмы. В заднем кармане я носил скребок, которым счищал тесто. Иногда большой кусок теста отлетал куда-то в сторону и залеплял все, на что попадал. Тесто снилось мне по ночам в кошмарных снах.

Я работал в производственной зоне. На другом конце пекарни занимались упаковкой и транспортировкой. Там проклятием были крошки. Они забирались всюду. Те, кто там работал, возвращались домой с крошками в волосах. Каждую смену часа два уходило на то, чтобы очистить от крошек механизмы. В задних карманах они носили маленькие щетки. Не сомневаюсь, что крошки снились им по ночам.

Почти с любой работой, за которую вам платят, связана какая-нибудь неприятная деталь. Если это не тесто или крошки, то, возможно, вы работаете на бритвенной фабрике и приходите домой с мелкими порезами на руках. Или вы работаете в Vmware, и вас посещают ночные кошмары, связанные с эмуляцией дефектов в сложных видеокартах, необходимых для компьютерных игр. Или вы разрабатываете Windows, и ваши страхи связаны с тем, что из-за какой-нибудь мелкой модификации перестанут работать миллионы программ и аппаратных устройств. Такая уж поганая особенность у вашей работы.

У нас в компании такой противной проблемой является выполнение FogBugz на собственных серверах наших клиентов. Джейсон Фрид (Jason Fried) из 37signals сделал хороший вывод, почему это занятие малоприятно (*www.37signals.com/svn/posts/724-ask-37signals-installable-software*): «Приходится сталкиваться с бесконечным разнообразием операционных сред, которое вне вашей власти. Когда возникает проблема, выяснить ее причину значительно труднее, если у вас нет доступа к ОС, или к каким-то посторонним программам, или к устройствам, мешающим установке, обновлению или обычной работе вашего программного продукта. Еще хуже положение при установке на удаленный сервер, когда приходится иметь дело с разными версиями Ruby, Rails, MYSQL и пр.». Джейсон заключает, что если бы они продавали инсталлируемые программы, то «определенно имели бы больше неприятностей». Да. Работа, которая доставляет вам неприятности, это то, что я имею в виду под «противными проблемами».

К несчастью, на рынке вам платят за решение противных, а не легких проблем. Как говорят йоркширские ребята, «где грязь, там и деньги».

Мы предлагаем оба вида FogBugz – с хостингом и инсталлируемый, и наши клиенты в четыре раза чаще выбирают установку на собственных сайтах. Инсталлируемая версия дает нам впятеро больший объем продаж. Она обходится нам дополнительно в одно-два жалования (стоимость технической поддержки). Кроме того, приходится использовать Wasabi[1], что имеет серьезные недостатки по сравнению с готовыми языками программирования, но это решение мы считаем наиболее экономичным и эффективным – с учетом наработанного кода – для поставки программного продукта, который можно инсталлировать в системах Windows, Linux и на Мак. С каким бы удовольствием я отказался от инсталлируемой версии FogBugz и делал все на наших серверах... У нас много стоек с прекрасными, хорошо администрируемыми серверами Dell с массой ресурсов, а техническая поддержка для версии с хостингом не стоит клиенту ничего. Насколько легче была бы наша жизнь! Но наши доходы настолько упали бы, что мы бы разорились.

Многие из нынешних симпатичных стартапов роднит между собой то, что у них есть простой маленький сайт, построенный на Ruby on Rails и Ajax, и им не нужно прилагать усилия для выхода на рынок, и они не ре-

[1] Закрытый внутрифирменный язык Fog Creek Software, диалект Basic с замыканиями, лямбда-функциями и активными записями типа Rails; компилируется в VBScript, JavaScript, PHP4 или PHP5. – *Прим. перев.*

шают никакие противные проблемы. Очень многие из этих компаний производят впечатление ненадежности и неуверенности, потому что в силу обстоятельств (вся компания – трое мальчишек и игуана) они пока не решили ни одной сложной задачи. А значит, они пока не решают и задачи, встающие перед другими. Люди платят за то, чтобы решали их проблемы.

Изготовить красиво выглядящее и простое в обращении приложение – тоже трудная задача, хотя здесь, как в балете: когда танцуют мастера, кажется, что им это дается без труда. Джейсон и 37signals вложили свои силы и сделали хороший проект, и теперь он приносит им деньги. Кажется, нет ничего проще, чем скопировать хороший проект, однако пример Microsoft, пытающейся скопировать iPod, демонстрирует обратное. Отличный проект – это решение противной проблемы, которое может обеспечить вам на редкость устойчивое конкурентное преимущество.

Вероятно, Джейсон правильно поступил, взявшись за противную проблему, для решения которой у него достаточно таланта (проектирование), потому что не похоже, чтобы для него это было сложно. Я программирую для Windows целую вечность, поэтому мне не кажется сложным написать с чистого листа на C++ программу установки FogBugz для Windows, решив при этом все эти противные проблемы с COM.

Единственный путь роста – для личности и для компании – это расширять границы того, что ты умеешь делать хорошо. В какой-то момент в 37signals могут прийти к выводу, что если нанять отдельного человека, который напишет скрипт для установки и станет осуществлять техническую поддержку инсталляций, то это хорошо окупится. Поэтому – если только это не вполне нормальное стремление сохранить небольшой размер компании – они смогут в конце концов преодолеть свое нежелание заниматься вещами, которые кажутся противными.

Может, это и не случится. Нет ничего дурного, чтобы ограничить свое дело только приятными вещами. Безусловно, этим и я иногда страдаю. Нет ничего дурного в том, чтобы определить для себя, что ты будешь решать только определенную группу задач в интересах небольшой избранной группы людей. Salesforce.com сумела достичь значительного роста, не выходя за рамки хостинга программного продукта. И есть множество маленьких и совершенно не стремящихся к росту программистских компаний, где уровень жизни сотрудников фантастически высок.

Но важно помнить, что как только вы решите еще одну противную проблему, ваш бизнес и рынок существенно вырастут. Хороший маркетинг, хороший проект, хорошая служба продаж, хорошее обслуживание и реше-

ние множества проблем, возникающих у ваших клиентов, – все это помогает одно другому. Сначала у вас появляется хороший проект, потом вы добавляете к нему несколько хороших функций и тем самым решаете проблемы своих клиентов, так что их число утраивается, а потом вкладываете некоторые усилия в маркетинг, и количество ваших клиентов снова утраивается, потому что гораздо больше людей узнает, что вы решаете их задачи, и тогда вы нанимаете людей для ведения торговли и еще раз утраиваете свою клиентскую базу, потому что теперь тем, кто знает о вашем решении, напоминают, чтобы они купили его, а затем вы добавляете функции, которые решают другие задачи для еще большего количества людей, и в конечном итоге у вас появляется возможность охватить своим программным продуктом столько людей, что можете считать, что вашими усилиями мир стал немного лучше.

P.S. Я не утверждаю, что 37signals увеличила бы объем своих продаж в пять раз, выпустив инсталлируемую версию Basecamp. Прежде всего, одна из причин, по которым мы продаем намного больше инсталлируемых версий FogBugz, состоит в том, что, как оказалось, покупатели считают их более дешевыми. (На самом деле, в долгосрочном плане – не дешевле, потому что приходится платить за сервер и самому его администрировать, но у каждого свое мнение.) Кроме того, мы установили такую низкую плату за поддержку инсталлируемой версии только потому, что 80% наших покупателей устанавливают ее на Windows Server. Из-за большого сходства систем семейства Windows нам гораздо проще осуществлять поддержку наибольшего общего знаменателя. Основные расходы на техническую поддержку вызваны для нас разнообразием существующих платформ UNIX – рискну предположить, что 20% наших продаж, приходящихся на UNIX, обуславливают 80% случаев, требующих технической поддержки. Если для инсталлируемой версии Basecamp потребуется UNIX, то стоимость поддержки непропорционально возрастет по сравнению с гипотетической инсталлируемой версией для Windows. И еще одна причина, по которой наш опыт может быть неприменим к 37signals. Мы продаем инсталлируемую версию уже семь лет, а версию с нашим хостингом – лишь около шести месяцев. У нас большая база инсталляций FogBugz у клиентов, привыкших к тому, что программа работает на их собственных серверах. Если считать только новых покупателей FogBugz, то отношение инсталлируемых версий к версиям с хостингом снизится до 3 к 1.

ЧАСТЬ ПЯТАЯ

Советы программистам

ГЛАВА ДВАДЦАТАЯ

Планирование с учетом прежних результатов (EBS)

26 октября 2007 года, пятница

Программист не очень любит составлять график работ (schedule). Обычно он пытается как-то этого избежать. «Как только – так сразу!» – говорит он, надеясь таким смелым и остроумным ответом вызвать улыбку на лице босса, чтобы тот, повеселев, забыл про график.

Если вы все же видите какие-то графики, то, в основном, сделанные без всякого энтузиазма. Хранят их в каком-нибудь забытом месте на файловом сервере. Когда, с опозданием на пару лет, команда все-таки выпускает продукт, приходит этот странный малый, у которого в кабинете стоит шкаф с папками, и подводит итог, читая старую распечатку, чем вызывает всеобщее веселье. «Надо же! Мы хотели за две недели переписать все на Ruby!»

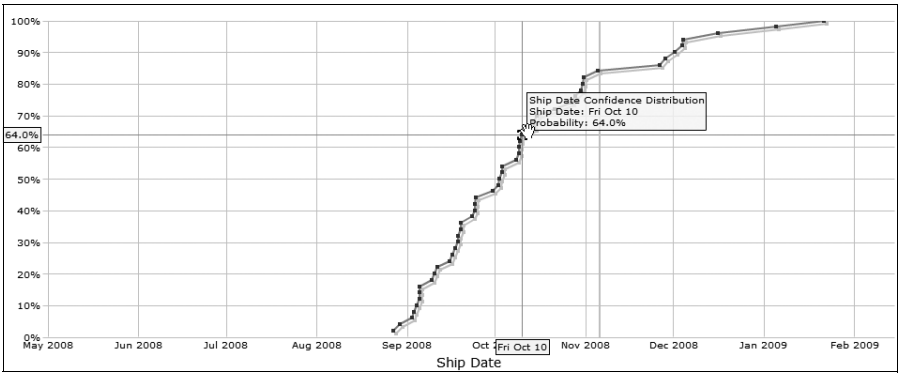
Очень смешно! Если, конечно, вы еще не обанкротились...

Нужно тратить время на то, что приносит максимум результатов на каждый вложенный доллар. И вы не сможете определить, во сколько долларов вам обойдется этот результат, если не будете знать, сколько времени займет ваша работа. Выбирая между реализацией «анимационной канцелярской скрепки» и «дополнительных финансовых функций», вы обязаны знать, сколько времени потребуется в каждом случае.

Почему программисты не хотят составлять график разработки? Тому есть две причины. Во-первых, график – это головная боль. Во-вторых, никто не верит, что график может быть реалистичным. Зачем мучиться с планированием, если на практике все будет по-другому?

В течение прошлого года или около того мы в Fog Creek разрабатывали систему, столь простую, что даже самые ворчливые из наших разработчи-

ков согласились ее применять. По нашему мнению, она позволяет создавать на редкость реалистичные планы проектов. Мы назвали ее Evidence-Based Scheduling (планирование с учетом прежних результатов), сокращенно EBS. Вы собираете *факты*, в основном из данных учета времени в прежних проектах, и учитываете их при составлении графика. В результате вы получаете не одну дату готовности продукта, а кривую вероятности завершения работы в каждый конкретный день. Это выглядит примерно так:



Чем круче кривая, тем больше уверенности в реальности даты поставки (Ship Date).

Вот как это делается.

1. Разбейте на куски

Глядя на график проекта, расписанный по дням или даже неделям, я знаю, что он не будет выполнен. Нужно разбить график на такие мелкие задачи, чтобы они измерялись в часах. Не более 16 часов каждая.

Это заставит вас реально определить, что вы намерены сделать. Написать такую-то подпрограмму. Создать вот это диалоговое окно. Организовать синтаксический разбор такого-то файла. Отдельные задачи разработки проще оценить, потому что вы когда-то уже писали подпрограммы, создавали диалоговые окна и анализировали файлы.

Если вы, не размениваясь по мелочам, небрежно определяете задачи на три недели вперед, – например «написать Ajax-редактор фотографий», –

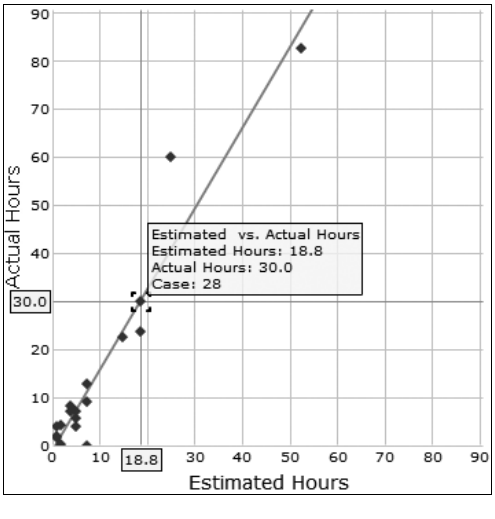
значит вы *не продумали, что будете делать*. В деталях. Шаг за шагом. А не продумав, что надо делать, вы не можете знать, сколько времени это у вас займет.

Установление 16-часового предела заставит вас *спроектировать* эту чертову штуку. Если вы отводите три недели на нечто неопределенное под названием «Ajax-редактор фотографий» без детального проекта, с глубоким прискорбием вынужден довести до вашего сведения, что вы обречены. Вы не обдумали шаги, необходимые для решения задачи, и наверняка пропустите многие из них.

2. Ведите учет затраченного времени

Сложно точно определить индивидуальные затраты времени. Как учесть отвлечения, непредсказуемые баги, совещание для обсуждения хода проекта или совершаемое раз в полгода жертвоприношение Windows, во время которого вам приходится целиком переустанавливать основную машину, на которой ведутся разработки? Да и без всего этого – как вы можете точно узнать, сколько времени займет реализация какой-либо подпрограммы? Да никак.

Так что ведите учет времени. Отслеживайте, сколько времени вы тратите на каждую задачу. Тогда позднее вы сможете сравнить реальное время со своей оценкой. Каждый разработчик должен собрать данные такого типа:



Каждая точка на диаграмме – это завершенная задача в координатах запланированного (Estimated Hours) и фактически потраченного (Actual Hours) времени. Поделив оценку на фактическое время, вы получите скорость: насколько быстро задача была решена в сравнении с планом. Со временем у вас накопятся массивы таких данных по каждому разработчику.

- Мифический *идеальный оценщик*, существующий только в нашем воображении, всегда точно определяет необходимое ему время. Его набор скоростей: {1; 1;1; 1; 1; ...}.
- Типичный *плохой оценщик* раскидает точки по всей координатной сетке, например вот так: {0,1; 0,5; 1,7; 0,2; 1,2; 0,9; 13,0}.
- Большинство исполнителей плохо определяют масштаб, но их относительные оценки правильны. Все задачи выполняются дольше, чем рассчитывалось, потому что оценка не учитывает исправление ошибок, совещаний комиссии, кофе-брейки и ненормального босса, который все время отвлекает. У этого *стандартного оценщика* скорости весьма постоянны, хотя и не дотягивают до 1, например: {0,6; 0,5; 0,6; 0,6; 0,5; 0,6; 0,7; 0,6}.

По мере приобретения опыта точность оценки растет. Так что можете смело выкинуть показатели, скажем, полугодовой давности.

Однако если в вашей команде появился новый разработчик, данных о скорости работы которого у вас нет, следует исходить из худшего варианта: назначьте ему какой-нибудь фиктивный набор скоростей с большим разбросом до тех пор, пока он не выполнит с полдюжины реальных задач.

3. Моделируйте будущее

Вместо того чтобы складывать оценки для получения точной даты поставки (что кажется правильным, но дает слишком неточные результаты), лучше смоделировать возможные исходы с помощью метода Монте-Карло. При этом вы берете 100 сценариев возможного развития событий, каждый из которых имеет вероятность 1%, и строите график вероятности готовности к поставке для любой даты.

Рассчитывая каждый возможный вариант развития событий для конкретного разработчика, нужно разделить оценку для каждой задачи на выбранную случайным образом скорость из данных по этому разработчику, собранных вами на шаге 2. Вот возможный пример:

Оценка:	4	8	2	8	16	
Случайная скорость:	0,6	0,5	0,6	0,6	0,5	**Сумма:**
Отношение:	**6,7**	**16**	**3,3**	**13,3**	**32**	**71,3**

Проделайте это 100 раз. Вероятность каждой суммы 1%, и теперь вы можете оценить вероятность поставки для каждой заданной даты.

А теперь посмотрите, что получилось.

- Для мифического идеального оценщика все скорости равны 1. Деление на 1 ничего не меняет. Поэтому все циклы моделирования дают одну и ту же дату окончания проекта, и ее вероятность 100%. Просто сказка!
- У плохого оценщика скорости разбросаны по всему пространству. Может встретиться и 0,1, и 13,0. Результаты всех циклов будут сильно различаться, потому что при делении на случайно выбранные значения скорости получатся весьма различные числа. Кривая распределения вероятности, которую вы получите, будет очень пологой, показывая равную вероятность завершения работ как завтра, так и в отдаленном будущем. Но это тоже ценная информация: она говорит, что предположительной дате поставки доверять нельзя.
- Стандартный оценщик располагает набором очень близких скоростей, например: {0,6; 0,5; 0,6; 0,6; 0,5; 0,6; 0,7; 0,6}. При делении на эти значения вы увеличиваете продолжительность выполнения задачи, поэтому в одной итерации вместо 8 часов получится 13, а в другой, скажем, 15. Это компенсирует неизменный оптимизм разработчика. Причем компенсирует *точно*, исходя из *доказанности и проверенности исторически зафиксированной меры его оптимизма*. И поскольку все эти зарегистрированные скорости весьма близки (их значение колеблется около 0,6), в каждом цикле моделирования вы получите довольно близкие значения, а в итоге – довольно узкий интервал для возможной даты поставки.

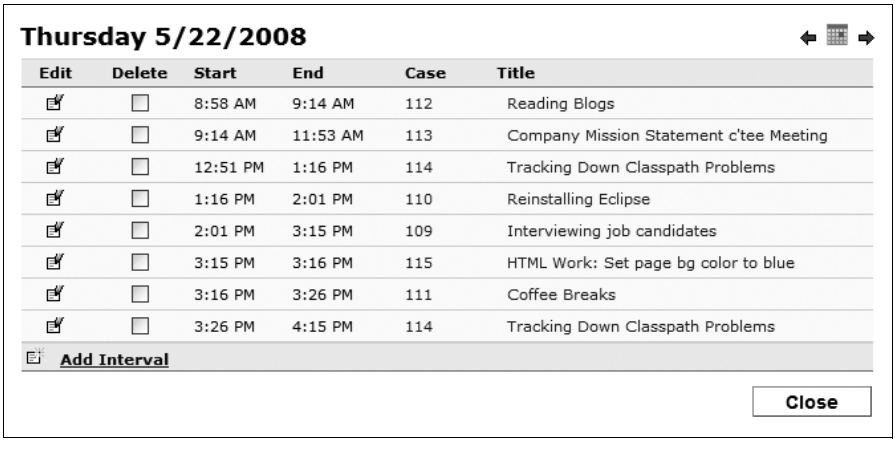
На каждом этапе моделирования по методу Монте-Карло вы, конечно, должны переводить данные из рабочих часов в рабочие дни, то есть нужно принять в расчет график работы каждого разработчика, его отпуск, праздники и так далее. И в каждом цикле вы должны посмотреть, кто из разработчиков завершит работу последним, потому что это и будет датой завер-

шения проекта командой в целом. Эти подсчеты довольно трудоемки, но, к счастью, как раз для такой работы существуют компьютеры.

Не нужно синдрома навязчивых состояний

Что делать с боссом, который постоянно отвлекает вас нескончаемыми рассказами о рыбалке? Или с совещаниями по продажам, на которые вы должны ходить, хотя делать вам там нечего? С чаепитиями? С необходимостью потратить полдня, помогая новому сотруднику установить нужные программы?

Когда мы с Бреттом разрабатывали технологию планирования в Fog Creek, нас сильно беспокоило, что есть отнимающие много времени вещи, предвидеть которые невозможно. Иногда они вместе съедают больше времени, чем уходит на написание кода. Может быть, для них сделать такие же оценки, ведя учет реального времени?



Thursday 5/22/2008

Edit	Delete	Start	End	Case	Title
		8:58 AM	9:14 AM	112	Reading Blogs
		9:14 AM	11:53 AM	113	Company Mission Statement c'tee Meeting
		12:51 PM	1:16 PM	114	Tracking Down Classpath Problems
		1:16 PM	2:01 PM	110	Reinstalling Eclipse
		2:01 PM	3:15 PM	109	Interviewing job candidates
		3:15 PM	3:16 PM	115	HTML Work: Set page bg color to blue
		3:16 PM	3:26 PM	111	Coffee Breaks
		3:26 PM	4:15 PM	114	Tracking Down Classpath Problems

Add Interval

Close

Можно поступить и так, если хотите. Система EBS сработает и в этом случае.

Но вам необязательно делать это.

Дело в том, что EBS настолько эффективна, что вам достаточно *не останавливать отсчет времени*, когда возникло отвлечение. Как ни странно, в этом случае EBS работает даже лучше.

Приведу короткий пример. Чтобы максимально его упростить, представим себе программиста Джона, который работает с хорошо предска-

зуемой скоростью и задача которого – писать однострочные функции чтения и записи значений на одном из этих недоразвитых языков программирования. Вот код, который он пишет изо дня в день:

```
private int width;
public int getWidth () { return width; }
public void setWidth (int _width} { width = _width; }
```

Знаю, знаю, это крайне тупой пример. Но нечто подобное наверняка встречалось вам и в реальной жизни.

Тем не менее. На каждую функцию у Джона уходит 2 часа. Таким образом, оценка длительности задач выглядит для него так:

{2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; ... }

Теперь представим, что у этого бедняги есть начальник, который время от времени затевает с ним двухчасовую беседу об особенностях ловли тунца. Конечно, Джон мог бы внести в свой график работы пункт «Тягомотина с тунцом» и зафиксировать потраченное на это время, но с его стороны это было бы неблагоразумно. Вместо этого Джон просто продолжает отсчет времени, и его график начинает выглядеть так:

{2; 2; 2; 2; 4; 2; 2; 2; 2; 4; 2; ... }

А показатели скорости разработки, соответственно, так:

{1; 1; 1; 1; 0,5; 1; 1; 1; 1; 0,5; 1; ... }

Теперь посмотрим, что происходит. При оценке методом Монте-Карло вероятность того, что оценка будет поделена на 0,5, *равна вероятности того, что босс прервет работу Джона над каждой из задач*. То есть EBS даст правильный график!

На самом деле, EBS гораздо точнее отражает всевозможные отвлечения, чем какой-нибудь одержимый учетом времени программист. *Вот почему она так эффективна*. Я объясняю этот факт следующим образом. Когда разработчика отвлекают от работы, он может поступить по-разному:

1. Поднять шум и потребовать зарегистрировать отвлечение в учете времени и в оценках, чтобы руководство знало, сколько времени отняли разговоры о рыбалке.
2. Поднять шум, что эти отвлечения не учитываются как рабочее время, если он не выполнил работу к положенному сроку, потому что ему не дают скорректировать его абсолютно точную оценку на вре-

мя дурацких разговоров о рыбалке, на которую его *даже не пригласили*.

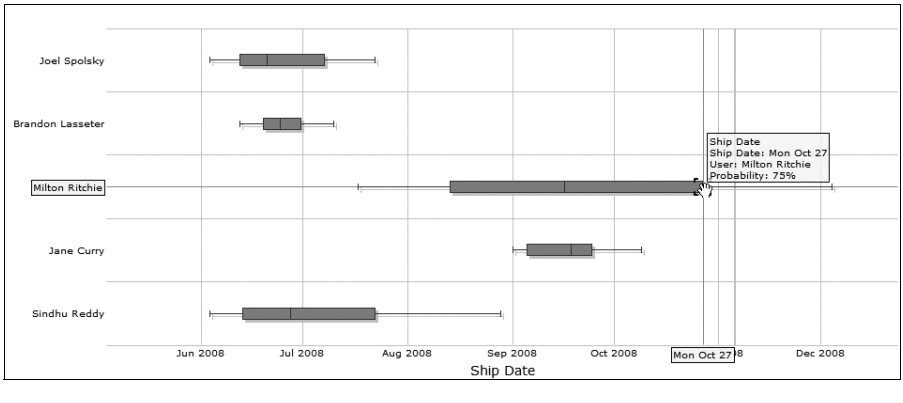
В любом случае, EBS предоставляет *одинаково точные результаты* независимо от того, пассивны или агрессивны ваши разработчики.

4. Активно управляйте своими проектами

Организовав такую систему, вы можете активно управлять своими проектами, добиваясь их своевременного завершения. Например, отсортировав функции по приоритетности, легко выяснить, как отказ от реализации маловажных функций может помочь своевременной поставке продукта.

Priority	50% Date
1 – Must Fix	**5/30/2008**
2 – Must Fix	**6/30/2008**
3 – Must Fix	**10/2/2008**
4 – Fix If Time	10/10/2008
5 – Fix If Time	10/19/2008
6 – Fix If Time	12/4/2008
7 – Don't Fix	3/1/2009

Можно также узнать распределение вероятных дат окончания работ *для каждого программиста*:



Некоторые разработчики (как Милтон на приведенной иллюстрации) могут создавать проблемы, поскольку дата окончания работ у них оказывается весьма неопределенной: им нужно учиться точнее планировать свою работу. Другие разработчики (как Джейн) очень точно определяют дату завершения своей работы, но она выходит за рамки предельного срока, поэтому придется снять с них часть задач. Другие разработчики (вот! это я!) не участвуют в критическом пути, и их можно оставить в покое.

Расползание границ проекта

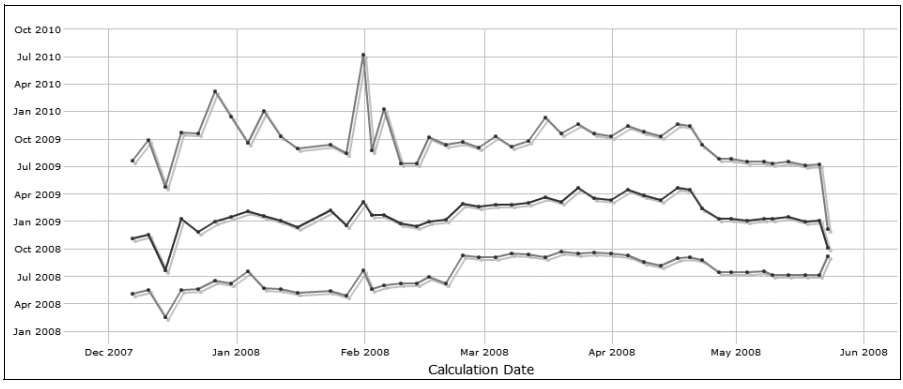
Если в начале работы вы спланировали все до последней детали, то EBS работает прекрасно. Но иногда добавляются изначально не запланированные функции. Это происходит, если у вас возникают новые идеи, или агенты по продажам начинают рекламировать то, чего нет в продукте, или кому-то из совета директоров пришла гениальная идея добавить электрокардиограф в ваше GPS-приложение для электромобиля, перевозящего игроков по площадке для гольфа. В результате возникают задержки, которые вы не могли предвидеть при создании первоначального графика.

В идеале, можно запланировать резервное время для разных случаев. Действительно, почему бы не ввести в график резерв для следующих нужд:

1. Новые функции.
2. Ответ на действия конкурентов.
3. Интеграция (настройка совместной работы кода разных разработчиков после его объединения).
4. Отладка.
5. Юзабилити-тестирование (и внесение по его итогам доработок в продукт).
6. Бета-тестирование.

Теперь, если понадобится добавить новые функции, можно выделить время для них из соответствующего резерва.

Как быть, если нужно добавить новые функции, а резервы исчерпаны? Дата окончания работ, рассчитываемая с помощью EBS, начнет запаздывать в сравнении с предельным сроком. Чтобы обнаружить это, нужно пересчитывать распределение вероятной даты окончания проекта в конце каждого рабочего дня:



По оси *X* откладывается дата проведения расчета, а по оси *Y* – дата готовности продукта к поставке. Три кривые – это графики: верхний – дата поставки с вероятностью 95%, средний – с вероятностью 50%, нижний – с вероятностью 5%. Чем ближе кривые друг к другу, тем меньше разброс вероятной даты поставки.

Если дата завершения работ отодвигается все дальше и дальше (кривая идет вверх), то у вас проблемы. Если каждый день задержка увеличивается больше чем на сутки, значит, вы наращиваете объем задач быстрее, чем решаете их, и так вы вообще никогда не завершите проект. Можно также узнать, сужается ли доверительный интервал для даты поставки (графики сближаются), что должно происходить, если ваш проект наверняка выходит на окончание к определенному сроку.

Дополнительные замечания

Вот еще несколько вещей, которые я выяснил за годы составления графиков работ:

1. **Только программист, занятый конкретной задачей, может оценить сроки ее выполнения.** Любая система, где график работ составляют менеджеры, спуская его программистам, обречена на провал. Только программист, который будет реализовывать функцию, может определить, какие действия ему при этом потребуются.
2. **Найденные ошибки нужно исправлять сразу, а время на исправление учитывать как потраченное на задачу.** Нельзя заранее планировать исправление ошибок, потому что вы не знаете, когда и ка-

кие ошибки обнаружатся. Время на исправление ошибок в новом коде – часть времени решения исходной задачи, которая была неправильно реализована. В этом случае с помощью EBS вы можете оценить время разработки полностью отлаженного, а не просто работающего, кода.

3. **Не позволяйте менеджерам требовать от разработчиков сокращенных оценок.** Многие неопытные менеджеры думают, что могут «мотивировать» программистов работать быстрее, дав им сжатые (нереально короткие) сроки. Я считаю такой способ мотивации просто глупостью. Отставая от графика, я чувствую себя убитым, подавленным и немотивированным. Работая с опережением, я чувствую себя бодрым и продуктивным. График – не место для психологических экспериментов.

   Почему же менеджеры делают такие попытки?

   В начале проекта технические менеджеры встречаются с представителями бизнеса и составляют список функций, реализация которых, как им *кажется*, потребует месяца три, хотя на самом деле уйдут все двенадцать. Когда вы думаете о том, что нужно бы написать код, но не обдумываете все этапы, всегда оказывается, что предполагаемое время n на практике превращается в $4n$. Составляя настоящий график, вы суммируете все задачи и обнаруживаете, что проект займет гораздо больше времени, чем предполагалось. Представителей бизнеса это не радует.

   Плохой менеджер пытается найти выход в том, чтобы заставить людей работать быстрее. Но это не жизненно. Можно набрать больше людей, но им нужно время, чтобы войти в курс дела, а до того они в течение нескольких месяцев будут работать в половину максимальных возможностей (и при этом снижать эффективность тех, кто будет их учить).

   Возможно, вам удастся *временно* выжать из своих людей на 10% больше сырого кода, но при этом за год они «выгорят» на 100%. Невелико достижение – все равно что рубить сук, на котором сидишь. И, конечно, когда люди перерабатывают, *удваивается* время на исправление ошибок, и проект лишь запаздывает еще больше. Отличная карма.

   Но что вам точно не удастся, так это сделать $4n$ из n, и если вы думаете, что это не так, пришлите мне, пожалуйста, код акций вашей компании – я сыграю на понижение.

4. **График – это коробка с деревянными брусками.** Если у вас есть несколько таких брусков, и они не помещаются в коробку, выхода два: взять коробку побольше или выкинуть несколько брусков. Если вы хотите завершить проект за 6 месяцев, но по графику получается 12, придется либо перенести срок завершения, либо отказаться от реализации каких-то функций. Сжать бруски нельзя, а если вам кажется, что можно, то вы просто обманываете себя и лишаетесь реальной возможности *заглянуть в будущее*.

Раз уж я заговорил об этом, отмечу, что реалистичное планирование хорошо еще и тем, что *вынуждает* вас избавляться от лишних функций. Почему это хорошо?

Предположим, вы задумали две функции. Одна из них действительно полезна и даст вашему продукту большие преимущества. Другая очень проста, и программистам не терпится начать писать ее код, хотя никакой пользы в ней нет.

Если не составить график работ, программисты сначала возьмутся за то, что делать легче/интереснее. При этом они выбьются из графика, и вам придется переносить срок поставки, чтобы реализовать полезную/важную функцию.

Если же составить график, то даже до начала работы вы поймете, что нужно что-то сократить, и сократите вы легкое/интересное, чтобы сделать полезное/важное. Заставив себя отказаться от маловажных функций, вы создадите более мощный, качественный продукт, с лучшим набором функций и за меньшее время.

Когда в далеком прошлом я работал над Excel 5, сначала у нас был громадный перечень подлежащих реализации функций, с которым мы намного превысили бы отведенные сроки. «Господи, – думали мы, – но они же все совершенно необходимы! Разве можно обойтись без помощника для редактирования макросов?»

Как выяснилось, выбора у нас не было, и для соблюдения сроков мы, как нам казалось, обглодали проект до костей. Все очень переживали из-за этого. Чтобы поднять настроение, мы объясняли себе, что не отказываемся от этих функций навсегда, а просто *переносим* их на следующую, шестую версию Excel.

Когда работа над пятой версией уже близилась к завершению, я и мой коллега Эрик Михельман (Eric Michelman) начали разработку спецификаций для шестой версии. Мы стали проверять список того, что было исклю-

чено из пятой версии в расчете включить это в шестую. И знаете, что? Это был совершенно идиотский список. *Ни одну* из тех функций не стоило реализовывать ни тогда, ни раньше. Мы очень правильно поступили, отобрав только те функции, которые позволяли нам уложиться в сроки. В ином случае Excel 5 писался бы вдвое дольше, а половина его функций была бы никому не нужна, но их поддержку – в целях обратной совместимости – пришлось бы осуществлять до скончания времен.

Заключение

Применять технологию «планирования с учетом прежних результатов» очень легко: нужно лишь в начале каждого этапа работы потратить один-два дня, сделав точные оценки, а потом ежедневно отмечать на графике время начала работы над очередной задачей, что отнимает несколько секунд. Зато выгода огромна: у вас будет реалистичный график работы.

Реалистичное планирование – ключ к созданию хороших программ. Оно заставляет вас начать с разработки самых важных функций и позволяет принимать правильные решения о том, что разрабатывать. В результате ваш продукт становится лучше, ваш босс счастлив, клиенты в восторге, а главное – вы сможете уходить домой ровно в пять.

P. S.

Система Evidence-Based Scheduling встроена в FogBugz 6.

ГЛАВА ДВАДЦАТЬ ПЕРВАЯ

Шестое письмо о стратегии

18 сентября 2007 года, вторник

IBM только что выпустила офисный пакет с открытым исходным кодом под названием IBM Lotus Symphony. Похоже на очередной дистрибутив StarOffice. Но я подозреваю, что они просто хотят вычеркнуть из памяти настоящий Lotus Symphony, в свое время объявленный чуть ли не вторым пришествием, но оказавшийся совершенно неудачным. Что-то вроде «Джильи» в мире программирования.

В конце 1980-х в Lotus ломали голову над тем, что же делать дальше с их флагманом электронных таблиц и графики Lotus 1-2-3. Было два очевидных пути. Во-первых, можно было добавить функции, например встроить текстовый редактор. Получившийся продукт был назван Symphony. Другим очевидным путем было сделать таблицу трехмерной. Так появился 1-2-3 версии 3.0.

В обоих случаях сразу возникала серьезная проблема старого DOS-ограничения на размер занимаемой памяти – 640 Кбайт. IBM начинала поставки компьютеров с чипами 80286, способными адресовать больше памяти, но в Lotus решили, что у программы, для которой нужен компьютер за 10 000 долларов, количество покупателей будет невелико. Поэтому они все ужимали и ужимали программу. Потратив 18 месяцев на то, чтобы втиснуть 1-2-3 для DOS в 640 Кбайт, они потеряли уйму времени и в конце концов отказались от трехмерности, чтобы таки запихнуть все в память. В случае же Symphony они просто крошили все функции налево и направо.

Ни один из путей не оказался верным. К моменту выпуска 1-2-3 версии 3.0 у всех уже были 80386-е машины с двумя или четырьмя мегабайтами

оперативной памяти. К тому же в Symphony были несовершенные таблицы, несовершенный текстовый редактор, да и вообще много несовершенного.

«Все это мило, старик, – скажете вы. – Но кому сегодня нужны старые программы в текстовом режиме?»

Потерпите еще немного, потому что история повторяется по трем разным направлениям, и самое правильное – ожидать тех же результатов.

Мало памяти и слабые процессоры

Испокон веков и примерно до 1989 года программистов крайне волновала эффективность. Тогда им просто не хватало памяти и процессорных тактов.

В конце 1990-х некоторые компании, включая Microsoft и Apple, обратили внимание (чуть раньше всех остальных), что закон Мура позволяет не очень сильно переживать из-за производительности и использования памяти. Делай что-нибудь крутое и жди, пока подоспеет железо! Microsoft выпустила первую версию Excel для Windows в то время, когда 80386-й был еще слишком дорог, но они были терпеливы. Через пару лет появился 80386SX, и каждый, кому по карману был клон за 1500 долларов, мог работать в Excel.

Благодаря резкому падению цен на память и ежегодному удвоению скорости процессоров у программиста появился выбор. Можно потратить полгода, переписывая внутренние циклы на Ассемблере, а можно те же полгода играть на ударных в рок-группе – в обоих случаях ваша программа будет работать быстрее. У программистов на Ассемблере нет толпы поклонниц.

В общем, нас теперь не сильно беспокоят проблемы производительности и оптимизации.

Кроме одного случая: исполнение JavaScript из Ajax-приложений броузерами. А так как в этом направлении движется практически все программирование, тут есть о чем подумать.

У многих из сегодняшних Ajax-приложений на клиентской стороне наберется больше мегабайта кода. На этот раз нас ограничивают не память и не такты процессора, а ширина канала и время компиляции. В любом случае, нужно сильно постараться, чтобы заставить прилично работать сложные Ajax-приложения.

Впрочем, история имеет свойство повторяться. Каналы становятся дешевле. Люди придумывают, как заранее скомпилировать JavaScript.

Разработчики, вложившие много труда в оптимизацию, заставляя приложения загружаться и работать быстро, в один прекрасный день обнаружат, что все их труды были в значительной мере напрасны или, по крайней мере, «не принесли конкурентного преимущества в долгосрочной перспективе», на языке экономистов.

Разработчики, которые ринулись добавлять в свои приложения крутые возможности, не заботясь о производительности, в конечном счете, предложат более удачные приложения.

Переносимый язык программирования

Язык программирования C был изобретен с явной целью облегчить перенос приложений с одного набора команд на другой. И он неплохо справился с этой задачей, но не был 100%-переносимым, в результате чего мы получили язык Java, более переносимый, чем C. Якобы.

Сейчас крупная проблема переносимости – это клиентский JavaScript, и особенно DOM в веб-броузерах. Писать приложение так, чтобы оно работало во всех броузерах, – сущий кошмар. Единственный выход – последовательно тестировать его на Firefox, IE6, IE7, Safari и Opera. И знаете, что? Мне некогда тестировать под Opera. Фигово быть Opera. У начинающих броузеров нет никаких шансов.

Что же теперь будет? Можете попросить Microsoft и Firefox лучше заботиться о совместимости. Удачи. Можете пойти по пути p-code/Java и соорудить небольшую песочницу поверх базовой системы. Но песочницы – сущее наказание: они медленные и противные, вот почему апплетов Java нет, нет, нет. Создавая песочницу, вы практически обрекаете себя на одну десятую скорости базовой платформы без возможности поддерживать все модные функции, которые есть на одной платформе и отсутствуют на других. (Я все еще жду, когда мне покажут такой апплет Java для телефона, который умеет работать со *всеми* функциями телефона, то есть с камерой, адресной книгой, SMS-сообщениями и GPS-приемником.)

Песочницы не работали тогда, не работают они и сейчас.

Что будет дальше? Победят те, кто сделает то же, что оказалось успешным в Bell Labs в 1978-м: создадут язык программирования, такой же переносимый и эффективный, как C. Он будет компилироваться в «родной» код (родным в данном случае будет JavaScript и DOM-реализации) с разными кодогенераторами для различных целевых платформ, а производительность будет волновать не вас, а разработчиков компиляторов. Он будет та-

ким же производительным, как обычный JavaScript, и иметь единообразный доступ к DOM, и он будет компилироваться в родной код для IE и родной код для Firefox автоматически и без проблем с переносимостью. Да, он заберется к вам в CSS и провернет все устрашающим, но все же корректным способом, поэтому вам никогда больше не придется думать о несовместимости CSS. Никогда. О, этот счастливый день!..

Интерактивность и стандарты интерфейсов пользователя

Интерфейс пользователя для мэйнфрейма IBM 360 назывался CICS – вы и сейчас можете увидеть его в аэропорту, если перегнетесь через стойку регистрации. Это зеленый экран размером 80×24, разумеется, работающий только в текстовом режиме. Мэйнфрейм отправляет форму «клиенту» (умному терминалу 3270). Умный он потому, что может отобразить форму и позволяет вам ввести в нее данные, не обращаясь при этом к самому мэйнфрейму. Это одна из причин, по которым мэйнфреймы были гораздо мощнее UNIX: процессор не участвовал во вводе вами строки, за это отвечал умный терминал. (Если вы не могли поставить всем умные терминалы, то покупали миникомпьютер System/1, подключаемый между терминалами и мэйнфреймом и управляющий вводом.)

Как бы то ни было, после заполнения формы вы нажимали кнопку ОТПРАВИТЬ, и все ваши ответы отправлялись на сервер для обработки. Потом машина присылала вам новую форму. И так снова и снова.

Ужас. Ну как сделать текстовый редактор для такой среды? (Вообще-то, никак. На мэйнфреймах никогда не было приличного текстового редактора.)

Это был первый этап. Он в точности соответствует эпохе HTML в Интернете. HTML – это CICS со шрифтами.

Второй этап начался, когда у каждого на столе появился персональный компьютер, и программисты внезапно получили возможность распихивать текст по всему экрану, где и когда угодно, и считывать каждое нажатие клавиши пользователем, что позволило делать симпатичные быстрые приложения, которым не требовалось ждать, пока вы нажмете кнопку ОТПРАВИТЬ, чтобы подключить к работе процессор. Например, можно было сделать текстовый редактор, автоматически переносящий слова на новую строку, когда текущая строка была заполнена до конца. Сразу. О, боже. Можете себе это представить?

Проблемой второго этапа было отсутствие четких стандартов для пользовательских интерфейсов. Программисты получили такую свободу, что каждый делал все, что хотел, и потому умение пользоваться программой *X* ничуть не облегчало вам работу с программой *Y*. В WordPerfect и Lotus 1-2-3 были совершенно разные системы меню, клавиатурные интерфейсы и структура команд. А о копировании данных между этими приложениями и речи не было.

То же происходит сегодня с Ajax-приложениями. Нет, конечно, пользоваться ими гораздо удобнее, чем первыми приложениями для DOS, – с тех пор мы все же кое-чему научились. Но Ajax-приложения бывают очень непохожими между собой и с трудом работают друг с другом – например, нельзя вырезать и вставлять объекты из одного Ajax-приложения в другое, поэтому я не знаю, как перетащить картинку из Gmail во Flickr. Эй, ребята, копирование и вставку изобрели четверть века назад!

Третий этап для персональных компьютеров – это Макинтош и Windows. Стандартизированный, единообразный пользовательский интерфейс с такими средствами, как множественные окна и буфер обмена, разработанные с учетом возможности совместной работы приложений. Благодаря новым графическим интерфейсам компьютер стал удобным и эффективным, что привело к взрыву популярности персональных компьютеров.

И если история повторяется, то можно ожидать какой-то стандартизации пользовательских интерфейсов Ajax, как это было в свое время с Microsoft Windows. Кто-нибудь напишет хороший SDK, позволяющий создавать мощные Ajax-приложения со стандартными элементами пользовательского интерфейса, способными работать совместно. И тот, чей SDK сумеет завоевать больше умов разработчиков, получит такое же прочное преимущество в конкурентной борьбе, как когда-то Microsoft с ее Windows API.

Если вы занимаетесь веб-разработками и не хотите поддерживать SDK, с которым работают все остальные, то обнаружите, что пользователям все меньше нравится работать с вашим веб-приложением, из-за того что оно не поддерживает копирование и вставку, синхронизацию адресной книги и еще какие-нибудь замысловатые функции обмена, которые в 2010 году окажутся нужными всем.

Представьте, например, что вы Google с его Gmail, и все у вас хорошо. Но вдруг появляется никому не известный проказник-стартап из Y Combinator и вызывает сумасшедший интерес, продавая NewSDK, сочетающий отличный переносимый язык программирования, компилирующийся в JavaScript, и, что еще лучше, огромную Ajax-библиотеку со всевозможны-

ми средствами организации взаимодействия между программами – не простыми копированием и вставкой, а мощными сложными функциями вроде синхронизации и единого управления личными данными (благодаря чему не придется сообщать обо всех своих делах Facebook и Twitter, можно все вводить в одном месте). Можете только посмеяться над ними, потому что их NewSDK занимает целых 232 мегабайта... 232 мегабайта JavaScript, и страница загружается за 76 секунд. Поэтому вашему приложению Gmail не грозит потеря пользователей.

Но пока вы сидите в своем googlекресле, попивая googlечино и чуть не лопаясь от самодовольства, выходят новые версии броузеров, которые поддерживают кэшированный скомпилированный JavaScript. И внезапно NewSDK становится очень быстрым. А Пол Грэм выдает им 6000 коробок быстрого супа, чтобы они еще три года не думали о еде, совершенствуя свой продукт.

А ваши программисты разводят руками: Gmail слишком велик, мы не можем перенести Gmail на этот дурацкий NewSDK. Нам придется переписать каждую строчку кода. Черт, да это же все надо писать заново, вся программная модель перевернута с ног на голову, она рекурсивная, и там столько скобок, сколько Google не купить. Последняя строка почти каждой функции состоит из 3296 закрывающих скобок. Чтобы их посчитать, придется купить специальный редактор.

А ребята, пишущие на NewSDK, выдают приличный текстовый редактор, приличный клиент электронной почты и убийственный публикатор событий для Facebook и Twitter, который синхронизируется с чем угодно, и люди начинают ими пользоваться.

И пока вы спали, все стали писать приложения на NewSDK, и они замечательные, и вдруг компаниям становятся нужны ТОЛЬКО приложения на NewSDK, а все эти классические чистые Ajax-приложения выглядят жалко, потому что они не умеют копировать и вставлять, синхронизировать и как следует работать друг с другом. И Gmail становится «историческим» приложением электронной почты, как WordPerfect в мире редакторов. Вы рассказываете детям, в каком были восторге от двухгигабайтного почтового ящика, а они смеются над вами. В их пилке для ногтей больше двух гигабайт.

Безумная история? Подставьте вместо «Google Gmail» «Lotus 1-2-3». NewSDK будет вторым пришествием Microsoft Windows, именно так Lotus потерял власть над рынком электронных таблиц. И это случится снова, на сей раз в Сети, потому что законы природы остаются прежними. Каких-то деталей мы можем не знать, но так будет.

ГЛАВА ДВАДЦАТЬ ВТОРАЯ

А ваш язык программирования такое умеет?

1 августа 2006 года, вторник

Однажды, просматривая свой код, вы замечаете два больших, почти одинаковых с виду блока. Они действительно совершенно одинаковы, за исключением того, что в одном упоминается Spaghetti, а в другом – Chocolate Mousse.

```
// Элементарный пример:

alert("I'd like some Spaghetti!");
alert("I'd like some Chocolate Mousse!");
```

(Все примеры написаны на JavaScript, но даже не зная JavaScript, можно понять суть.)

Дублировать код, как вы знаете, нехорошо, поэтому вы создаете функцию:

```
function SwedishChef( food )
{
  alert("I'd like some " + food + "!");
}

SwedishChef("Spaghetti");
SwedishChef("Chocolate Mousse");
```

Ладно, это простейший пример, но вы можете представить, что он гораздо сложнее. Этот код лучше по многим причинам, о которых вы слышали миллион раз. Сопровождаемость, Читабельность, Абстракция = Хорошо!

Теперь вы замечаете еще два блока кода, которые выглядят почти одинаково, за исключением того, что один несколько раз вызывает функцию BoomBoom, а другой – функцию PutInPot. В остальном код почти одинаковый.

```
alert("get the lobster");
PutInPot("lobster");
PutInPot("water");

alert("get the chicken");
BoomBoom("chicken");
BoomBoom("coconut");
```

Теперь вам нужен способ передать в функцию аргумент, который сам является функцией. Это очень важная возможность, потому что она увеличивает ваши шансы найти одинаковый код, который можно вынести в отдельную функцию.

```
function Cook( i1, i2, f )
{
  alert("get the " + i1);
  f(i1);
  f(i2);
}

Cook("lobster", "water", PutInPot );
Cook("chicken", "coconut", BoomBoom );
```

Ура! Мы передали функцию как аргумент.

А ваш язык программирования такое умеет?

Подождите... предположим, вы еще не определили функции PutInPot и BoomBoom. Не будет ли изящнее написать их прямо в вызове, вместо того чтобы объявлять где-то в другом месте?

```
Cook( "lobster",
    "water",
    function(x) { alert("pot " + x); }  );
Cook( "chicken",
    "coconut",
    function(x) { alert("boom " + x); } );
```

Черт возьми, это удобно. Заметьте, я создаю здесь функцию на лету, даже не беспокоясь о том, чтобы дать ей имя, – просто подымаю ее за уши и опускаю в функцию.

Как только вы привыкнете к идее безымянных функций как аргументов, вы начнете повсюду замечать код, который, скажем, делает что-то с каждым элементом массива.

```
var a = [1,2,3];

for (i=0; i<a.length; i++)
{
  a[i] = a[i] * 2;
}

for (i=0; i<a.length; i++)
{
  alert(a[i]);
}
```

Очень часто возникает необходимость что-то сделать с каждым элементом массива, поэтому можно написать функцию, которая выполнит эту работу вместо вас:

```
function map(fn, a)
{
  for (i = 0; i < a.length; i++)
  {
    a[i] = fn(a[i]);
  }
}
```

Теперь можно переписать предыдущий код:

```
map( function(x){return x*2;}, a );
map( alert, a );
```

Другая распространенная операция с массивом – какое-нибудь объединяющее действие со всеми его элементами.

```
function sum(a)
{
  var s = 0;
  for (i = 0; i < a.length; i++)
    s += a[i];
  return s;
}

function join(a)
```

```
{
   var s = "";
   for (i = 0; i < a.length; i++)
      s += a[i];
   return s;
}

  alert(sum([1,2,3]));
  alert(join(["a","b","c"]));
```

Функции sum и join выглядят так похоже, что возникает желание абстрагировать их сущность в общую функцию, объединяющую элементы массива в одно значение:

```
function reduce(fn, a, init)
{
   var s = init;
   for (i = 0; i < a.length; i++)
      s = fn( s, a[i] );
   return s;
}

function sum(a)
{
   return reduce( function(a, b){ return a + b; },
              a, 0 );
}

function join(a)
{
   return reduce( function(a, b){ return a + b; },
              a, "" );
}
```

Многие старые языки программирования просто не позволяют проделывать такие трюки. Другие языки позволят вам сделать это, но с трудом (например, в C есть указатели на функции, но вы должны объявить и определить функцию где-либо в другом месте). Объектно-ориентированные языки программирования еще не вполне уверены, что стоит разрешить вам делать что-нибудь с функциями.

Java требует от вас создать отдельный объект с одним методом, так называемый функтор, если вы хотите представить функцию как объект первого рода. Добавьте к этому тот факт, что многие ОО-языки требуют соз-

давать отдельный файл для каждого класса, от чего код быстро делается неуклюжим. Если ваш язык программирования требует применения функторов, у вас не будет всех преимуществ современной среды программирования. Узнайте, не возвращают ли они деньги.

Много ли вы выиграете от возможности писать крошечные функции, которые всего лишь пробегают по массиву, проделывая что-нибудь с каждым элементом?

Что ж, вернемся к функции `map`. Когда вам нужно поочередно проделать что-нибудь с каждым элементом, очень часто порядок не имеет значения. Вы же можете пройти массив в прямом или обратном направлении, и результат будет одинаковым, верно? В действительности, если в вашем распоряжении есть два процессора, можно написать какой-нибудь код для того, чтобы каждый процессор обрабатывал половину элементов, и тогда `map` вдруг станет в два раза быстрее.

А может, чисто гипотетически, у вас сотни тысяч серверов в нескольких центрах обработки данных, разбросанных по всему миру, и действительно большой массив, содержащий, скажем, опять чисто гипотетически, все содержимое Интернета. Теперь вы можете запустить `map` на тысячах компьютеров, каждый из которых будет решать маленькую часть задачи.

Таким образом, сегодня написание, к примеру, какого-то по-настоящему быстрого кода для поиска во всем содержимом Интернета просто сводится к вызову функции `map` с простой функцией поиска в строке в качестве аргумента.

Самое интересное, на что я хочу обратить здесь ваше внимание, это что если представить себе, что `map` и `reduce` – функции, которыми могут пользоваться и пользуются все, то вам нужен только супергений, который напишет сложный код для запуска `map` и `reduce` на глобальном параллельном массиве компьютеров, и весь старый код, который отлично работал, когда вы запускали небольшие циклы, по-прежнему будет работать, только во много раз быстрее, а значит, позволит мгновенно решать действительно большие задачи.

Позвольте повториться. Абстрагируя концепцию циклов, вы можете обрабатывать их любым желаемым образом, включая такую реализацию, которая хорошо масштабируется при появлении дополнительного «железа».

Теперь вам понятно мое недавнее недовольство по поводу того, что студентов не учат ничему, кроме Java:

Тот, кто не понимает функционального программирования, не смог бы изобрести MapReduce – алгоритм, благодаря которому Google работает с гигантскими объемами данных, размещенных на многочисленных серверах. Термины «Map» и «Reduce» пришли из Lisp и функционального программирования. Понять, как работает MapReduce, может всякий, кто помнит из своего курса 6.001 или аналогичного ему, что у чисто функциональных программ нет побочных эффектов, поэтому их просто распараллеливать. Тот факт, что в Google смогли придумать MapReduce, а в Microsoft – нет, отчасти объясняет, почему Microsoft по-прежнему отстает в своих попытках наладить базовые функции поиска, тогда как Google уже решает новую задачу – построить Skynet^H^H^H^H^H^H, крупнейший в мире суперкомпьютер с массовым параллелизмом. Я не уверен, что в Microsoft в достаточной мере понимают, как сильно они отстали в этом отношении. (глава 8)

ОК. Надеюсь, теперь я вас убедил, что языки программирования, где функции – это объекты первого рода, предоставляют больше возможностей для абстрагирования, благодаря чему можно уменьшить объем кода, сделать код более надежным, облегчить его многократное использование и масштабирование. Множество приложений Google используют MapReduce, и все они выигрывают, когда кто-нибудь оптимизирует его или исправляет ошибку.

А сейчас я войду в раж и стану доказывать, что наиболее эффективны среды программирования, позволяющие работать *на разных уровнях абстракции*. Паршивый старый FORTRAN даже не позволял создавать функции. В языке C есть указатели на функции, но они ооочень уродливы, не могут быть безымянными, и их реализация должна помещаться не там, где они используются. Java заставляет использовать функторы, которые еще уродливее. Как заметил Стив Егге (Steve Yegge), Java – это Королевство существительных (*steve-yegge.blogspot.com/2006/03/execution-in-kingdom-of-nouns.html*).

Примечание

Я не использовал FORTRAN уже 27 лет. Конечно, в нем есть функции. Должно быть, я имел в виду GW-BASIC.

ГЛАВА ДВАДЦАТЬ ТРЕТЬЯ

Как заставить неправильный код выглядеть неправильно

11 мая 2005 года, среда

Давным-давно, в сентябре 1983-го я поступил на свою первую настоящую работу в Oranim, крупный израильский хлебозавод, каждую ночь выпекающий около 100 000 буханок хлеба в шести гигантских печах размером с авианосец.

Прежде всего в пекарне меня поразила грязь. Пожелтевшие печи, ржавые машины, всюду смазка.

Я спросил: «Здесь всегда так грязно?»

«О чем это ты? – сказал менеджер. – Уборка только что закончилась. Здесь уже много недель не было так чисто».

Боже милостивый.

Два месяца убирая пекарню по утрам, я понял, что они имели в виду. Чистота в пекарне означала, что на машинах нет теста. И что нет гниющего теста в мусорных бачках. И что тесто не валяется на полу.

Чистота не означала, что печи выкрашены свежей белой краской. Их красили, но раз в десять лет, а не каждый день. Чистота не означала, что нигде не видно смазки. Там было много машин, которые нужно было регулярно смазывать, и тонкий слой чистого масла обычно служил признаком того, что машину только что почистили.

Что такое чистота в пекарне, нужно было понять. Зайдя со стороны, невозможно решить, чисто в этом месте или нет. Постороннему в голову не придет осмотреть внутренние поверхности формовщика (машина, которая скатывает квадратные куски теста в круглые) и определить, хорошо ли

они отскоблены. Посторонний обратил бы внимание на выцветшие панели старой печи, потому что они *огромны*. Но пекаря меньше всего волнует, что краска на внешних панелях печи начала немного желтеть. На вкусе хлеба это не отражается.

После двух месяцев в пекарне вы бы узнали, что такое «чисто».

То же самое в программировании.

Если вы начинающий программист или пробуете читать код на новом языке, все выглядит одинаково непостижимо. Пока вы не освоите язык, вы не заметите даже очевидные синтаксические ошибки.

На первом этапе обучения вы уже начинаете обращать внимание на то, что принято называть «стилем кодирования». И вы замечаете, что в коде не соблюдаются стандарты отступов или применения заглавных букв в именах переменных.

В такой момент обычно говорят: «Лопни моя селезенка! Мы должны установить какие-то правила для единообразного кодирования!» – и тратят весь следующий день на составление правил написания кода для своей команды, остаток недели – на споры о Едином Истинном Стиле Расстановки Скобок, еще три недели – на переписывание старого кода в соответствии с Единым Истинным Стилем Расстановки Скобок, пока менеджер не засечет вас, сделав выговор за напрасную трату времени, не приносящую никакой прибыли, и вы решите, что вполне можно переформатировать код позже, в результате только половина вашего кода соответствует Истинному Стилю Расстановки Скобок, но скоро вы об этом забудете и увлечетесь чем-то еще, столь же бесполезным в смысле прибыли, например станете заменять один строковый класс на другой.

По мере совершенствования в написании кода в определенной среде разработки вы учитесь видеть и нечто другое. То, что совершенно законно и правильно с точки зрения правил кодирования, но вызывает беспокойство.

Например, в C:

```
char* dest, src;
```

Это абсолютно верный код; он может соответствовать вашему соглашению о кодировании и даже правильно делать то, для чего предназначен, но при наличии достаточного опыта работы на C вы заметите, что здесь dest объявляется как указатель на char, в то время как src объявляется просто как char, и возможно, это не то, чего вы хотели. Этот код выглядит неопрятно.

Еще более тонкий случай:

```
if (i != 0)
  foo(i);
```

В этом случае код правилен на 100%, соответствует большинству соглашений о кодировании и в нем нет ничего плохого, но тот факт, что тело оператора if, содержащего один оператор, не помещено в фигурные скобки, может вызвать беспокойство, потому что возникает мысль, не угораздит ли кого-нибудь вставить туда еще одну строку кода:

```
if (i != 0)
  bar(i);
  foo(i);
```

и забыть добавить фигурные скобки, что ненароком заставит foo(i) выполняться безусловно! То есть при виде блоков кода, не заключенных в фигурные скобки, вас начинает беспокоить крошечное, микроскопическое, мизерное ощущение неопрятности.

Итак, пока я упомянул три уровня квалификации программиста:

1. Вы не отличаете чистое от грязного.
2. У вас есть поверхностное представление о чистоте, главным образом на уровне соответствия соглашениям о кодировании.
3. Вы начинаете улавливать тонкие намеки на скрытую неопрятность, и они раздражают вас достаточно сильно для того, чтобы вы исправили код.

Однако есть еще один, более высокий уровень, о котором я и хочу поговорить:

4. Вы специально строите код так, чтобы ваше чутье на неопрятность эффективнее помогало избавить код от ошибок.

Сделать код надежным, *изобретая правила* написания кода, чтобы ошибки бросались в глаза, – настоящее искусство.

Сейчас мы рассмотрим небольшой пример, затем я покажу общее правило, которое поможет вам изобретать эти самые соглашения для написания надежного кода, что в конце концов оправдает один из видов «венгерской нотации», не столь уж тошнотворный, и послужит критикой применения обработки исключений в определенных обстоятельствах, хотя, скорее всего, вы не назовете эти обстоятельства типичными.

Но если вы твердо уверены, что венгерская нотация – это плохо, а обработка исключений – лучшее изобретение человечества после шоколадно-молочного коктейля, и других мнений даже слышать не хотите, то вместо

этого, к примеру, посмотрите классные комиксы (*www.neopoleon.com/home/blogs/neo/archive/2005/04/29/15699.aspx*) – может, вы ничего и не потеряете. Немного ниже я начну разбирать реальные примеры кода, от чего вы, скорее всего, заснете раньше, чем успеете возмутиться. Да. Думаю, вас нужно убаюкать, и пока вы дремлете и не станете активно сопротивляться, незаметно протолкнуть идею, что венгерская нотация = хорошо, а исключения = плохо.

Пример

Правильно. Начнем с примера. Предположим, что вы разрабатываете некое веб-приложение (потому что на них, кажется, все сегодня помешались).

Значит так. Существует такой вид уязвимости компьютерных систем, как межсайтовый скриптинг, или XSS. Не буду вдаваться в детали: достаточно знать, что в целях безопасности ваше веб-приложение никогда не должно передавать обратно строки, введенные пользователем в формах.

Например, если у вас есть веб-страница с полем ввода для ответа на вопрос «Как вас зовут?», а затем эта страница пересылает вас на другую страницу, где сказано «Привет, Элмер!» (если пользователь ввел имя Элмер), ну, в общем, это опасная уязвимость, потому что пользователь мог ввести везде вместо «Элмер» любой код HTML или JavaScript, и этот вредный код JavaScript может сделать какую-нибудь гадость, и теперь эти гадости, похоже, будут исходить от вас, например они могут прочитать ваши cookie и послать информацию из них на злой сайт доктора Зло.

Давайте запишем это в виде псевдокода. Представьте, что

```
s = Request("name")
```

читает введенные данные (аргумент POST) из HTML-формы. Если вы где-то поместите код:

```
Write "Hello, " & Request("name")
```

то ваш сайт становится уязвимым для XSS-атак. Этого достаточно.

Вместо этого вы должны перекодировать строку, прежде чем отправить ее обратно в HTML. Перекодировка означает замену символов: " – на ", > – на > и так далее. Следующий код является совершенно безопасным:

```
Write "Hello, " & Encode(Request("name"))
```

Все строки, которые приходят от пользователя, опасны. Любую опасную строку не следует выводить без перекодирования.

Попробуем придумать соглашение о кодировании, гарантирующее, что если вы когда-либо сделаете эту ошибку, код сразу же станет выглядеть неправильно. А если код неправильно выглядит, то разработчик или тот, кто просматривает код, сможет отловить ошибку.

Решение № 1

Одно из решений состоит в том, чтобы кодировать любую строку сразу, как только она пришла от пользователя:

```
s = Encode(Request("name"))
```

Таким образом, наше соглашение говорит: если вы увидите Request, который не заключен в Encode, то код неправильный.

Начинайте тренировать зрение на поиск голых Request, потому что они нарушают соглашение.

Такой способ действует – в том смысле, что, следуя этому соглашению, вы избежите XSS-атак, но не факт, что такая архитектура – лучшая. Например, если вам нужно хранить эти пользовательские строки где-нибудь в базе данных, нет смысла записывать их в базу в HTML-кодировке, потому что они могут предназначаться не для HTML-страницы, а, например, для приложения, обрабатывающего кредитные карточки, которое собьется, если получит строку в HTML-кодировке. Большинство веб-приложений разрабатывают по принципу, чтобы не перекодировать все строки внутри до *самого последнего момента* перед их отправкой на HTML-страницу, и это, по всей видимости, верная архитектура.

Итак, нам нужно какое-то время хранить данные в опасном формате.

Хорошо. Попробую еще раз.

Решение №2

А если условиться, что все строки должны кодироваться при выводе?

```
s = Request("name")

// намного дальше:
Write Encode(s)
```

Теперь всякий раз, видя голый Write без Encode, вы догадаетесь, что здесь что-то упущено.

Однако такой метод не всегда действует правильно. Иногда в коде встречаются фрагменты HTML, которые *нельзя* перекодировать:

```
If mode = "linebreak" Then prefix = "<br>"
// намного дальше:
Write prefix
```

Это не соответствует нашему соглашению, которое требует, чтобы мы перекодировали строки при выводе:

```
Write Encode(prefix)
```

Но теперь префикс "
", которым должна начаться новая строка, будет перекодирован в
 и пользователь увидит символы < b r >. Опять неправильно.

Итак, иногда нельзя перекодировать строку при вводе, а иногда – при выводе, таким образом, ни одно из предложенных соглашений не годится. А без соглашения мы постоянно подвергаемся риску написать примерно такой код:

```
s = Request("name")
…через несколько страниц…
name = s
…через несколько страниц…
recordset("name") = name // сохранить имя в БД в поле "name"
…через несколько дней…
theName = recordset("name")
…через несколько страниц или месяцев…
Write theName
```

Мы не забыли перекодировать строку? Неясно, как можно заметить ошибку. Непонятно, где копать. Если такого кода много, уйму сил придется потратить на то, чтобы проследить происхождение каждой строки, которую нужно вывести, и удостовериться, что она была перекодирована.

Правильное решение

Теперь позвольте мне предложить соглашение о кодировании, которое работает. У нас будет только одно правило:

Все строки, которые приходят от пользователя, должны быть сохранены в переменных (или полях базы данных), имена которых начинаются с префикса us (Unsafe String – опасная строка). Все строки, перекодиро-

ванные для HTML или полученные из надежного источника, должны быть сохранены в переменных, имена которых начинаются с префикса s (Safe – безопасная строка).

Позвольте мне переписать тот же самый код, изменяя только имена переменных, так чтобы это соответствовало нашему новому соглашению.

```
us = Request("name")
…через несколько страниц…
usName = us
…через несколько страниц…
recordset("name") = usName // сохранить имя в БД в поле "name"
…через несколько дней…
sName = Encode(recordset("name"))
…через несколько страниц или месяцев…
Write sName
```

Хочу обратить ваше внимание на то, что если вы, соблюдая новое соглашение, допустили ошибку при действиях с опасной строкой, это всегда можно обнаружить *в одной-единственной строке кода*:

```
s = Request("name")
```

Заведомо неправильно, потому что результат Request присваивается переменной, название которой начинается с s, а это противоречит правилам. Результат Request всегда опасен, поэтому он должен всегда присваиваться переменной, имя которой начинается us.

Всегда правильно:

```
us = Request("name")
```

Всегда правильно:

```
usName = us
```

Явно неправильно:

```
sName = us
```

Правильно:

```
sName = Encode(us)
```

Явно неправильно:

```
Write usName
```

Правильно:

```
Write sName
```

Тоже правильно:

```
Write Encode(usName)
```

Каждую строку кода можно просмотреть *отдельно*, и если каждая строка кода правильна, то весь код верен в полном объеме.

Пользуясь таким соглашением о кодировании, нужно натренироваться находить в коде Write us*XXX*, и вы сразу заметите ошибку, зная, как ее исправить. Сначала будет нелегко увидеть неправильный код, но после трех недель такой работы ваши глаза научатся это делать – так же, как рабочие хлебопекарни научились, бросив один взгляд, сразу замечать: «Так, никто не почистил формовщик внутри! Что за бездельники тут работали?»

На самом деле, можно немного развить правило и переименовать (или обернуть) функции Request и Encode так, чтобы они стали UsRequest и Sencode. Иными словами, имена функций, возвращающих опасные или безопасные строки, будут начинаться с Us и S, точно так же, как и переменные. Теперь взгляните на код:

```
us = UsRequest("name")
usName = us
recordset("usName") = usName
sName = Sencode(recordset("usName"))
Write sName
```

Видите, что я сделал? Теперь сразу видно ошибку, когда кодовые строки слева и справа от знака присваивания начинаются с разных префиксов.

```
us = UsRequest("name")  // порядок, слева и справа начинается с "us"
s = UsRequest("name")   // ошибка
usName = us             // порядок
sName = us              // явно неправильно
sName = SEncode(us)     // явно правильно
```

Пожалуй, можно сделать еще один шаг и переименовать Write в WriteS, а SEncode – в SFromUs:

```
us = UsRequest("name")
usName = us
recordset("usName") = usName
```

```
sName = SfromUs(recordset("usName"))
WriteS sName
```

В результате ошибки станут еще заметнее. Ваши глаза научатся обнаруживать код «с душком», и это поможет вам находить неясные ошибки защиты в процессе обычного написания или чтения кода.

Сделать так, чтобы плохой код бросался в глаза, – прекрасный подход, но не для всех проблем безопасности такое решение – лучшее. Не все ошибки или недостатки можно так обнаружить, потому что трудно просмотреть каждую строку кода. Но это явно лучше, чем ничего, и я с удовольствием приму соглашение о кодировании, при соблюдении которого неправильный код, по крайней мере, выглядит неправильно. Оно приносит пользу каждый раз, когда программист пробегает глазами строку кода для проверки отсутствия некоторой конкретной ошибки.

Общее правило

Для успешного применения правил, благодаря которым неправильный код и выглядит неправильно, нужно, чтобы все необходимое на экране располагалось как можно ближе друг к другу. Когда я ищу ошибки в коде и вижу строку, то где бы она ни находилась, мне нужно знать, опасная она или нет. И я не хочу, чтобы выяснить это, лезть в другой файл или на другую страницу. Я хочу видеть это тут же *на месте*, а для этого требуется соглашение об именовании переменных.

Есть ряд примеров, когда можно улучшить код, помещая нужные вещи близко друг к другу. Большинство соглашений о кодировании содержит правила наподобие следующих:

- Функции должны быть короткими.
- Объявляйте переменные как можно ближе к тому месту, где вы будете их использовать.
- Не создавайте персональные языки программирования с помощью макросов.
- Не применяйте `goto`.
- Закрывающая фигурная скобка должна быть на одном экране с соответствующей ей открывающей скобкой.

Общее в этих правилах то, что все они стремятся как можно компактнее разместить всю информацию, относящуюся к тому, чем занимается не-

кая строка кода. Это повышает вероятность того, что ваши глаза смогут охватить все, что нужно для понимания.

Должен признаться, я слегка опасаюсь особенностей языка, позволяющих что-то скрыть. Когда вы видите код

```
i = j * 5;
```

то в языке C вы, по крайней мере, знаете, что j умножается на 5, и результаты сохраняются в i.

Но если вы видите тот же фрагмент в C++, то не знаете о нем ничего. Абсолютно ничего. В C++ единственный способ узнать, что здесь действительно происходит, это выяснить, какие типы имеют i и j, а они могут быть объявлены в совершенно другом месте. Дело в том, что в зависимости от типа j оператор * может оказаться перегруженным для выполнения какой-нибудь заумной операции вместо обычного умножения. Да и оператор = может оказаться перегруженным для типа i, а типы могут оказаться несовместимыми, и в результате может быть автоматически вызвана функция приведения типов. И единственный способ разобраться во всем этом – не просто проверять типы, но отыскивать код, реализующий именно эти типы, и вам остается только молиться, встретив наследование, потому что тогда вам придется тащиться вверх по иерархии классов и искать, где все-таки на самом деле находится нужный код, а если где-нибудь есть полиморфизм, то вы *действительно* попали в беду, потому что тогда недостаточно знать, с какими типами *объявлены* i и j, – надо точно знать, какие типы у них *именно сейчас*, а сколько кода придется для этого просмотреть, уже вовсе неизвестно, и нет уверенности, что вы просмотрели все, потому что существует «проблема остановки» (уф!).

Видя в C++ код i = j * 5, вы можете рассчитывать только на себя, ребята, и, по-моему, это уменьшает возможность обнаружить потенциальные проблемы, просто глядя на код.

Конечно, никто не собирался осложнять вам жизнь. Если вам позволяют делать всякие умные штуки вроде перегрузки оператора *, так это для того, чтобы помочь вам обеспечить хорошую надежную абстракцию. Черт возьми, j имеет тип Unicode String, а умножение строки Unicode на целое число – *несомненно,* прекрасная абстракция для преобразования Традиционного Китайского в Стандартный Китайский, не так ли?

Беда в том, что надежных абстракций не бывает. Я уже достаточно много говорил об этом в Законе Дырявых Абстракций в другой книге, поэтому не стану повторяться.

Скотт Мейерс (Scott Meyers)[1] сделал себе имя, демонстрируя разнообразные ловушки, подстерегающие вас в абстракциях, по крайней мере, в C++.

Так. Я теряю нить. Пожалуй, подытожу все сказанное выше:

Ищите соглашения о кодировании, заставляющие неправильный код выглядеть неправильно. Старайтесь делать так, чтобы вся нужная информация размещалась в одном и том же месте экрана, так, чтобы ваш код позволял ясно видеть определенные проблемы и сразу же их устранять.

Я – венгр

И так, вернемся к пресловутой венгерской нотации.

Венгерская нотация была изобретена программистом Microsoft Чарльзом Симони (Charles Simonyi). Одним из крупных проектов, над которыми работал Симони в Microsoft, был Word; на самом деле, он руководил проектом по созданию первого в мире текстового редактора, работающего в режиме WYSIWYG (того, что в Xerox Parc называли Bravo).

При обработке текстов в режиме WYSIWYG окна прокручиваются, поэтому все координаты можно интерпретировать как относительно окна, так и относительно страницы, а это большая разница, и очень важно корректно работать с ними.

Это, как я предполагаю, и стало одной из веских причин, по которым Симони начал применять то, что впоследствии было названо «венгерской нотацией». Она была похожа на венгерский язык (Симони – венгр), отсюда и пошло название. В версии венгерской нотации, предложенной Симони, имя каждой переменной содержало префикс из строчных букв, указывающий вид данных, которые содержит эта переменная.

Я нарочно использую здесь слово «вид» (kind), потому что Симони по ошибке использовал в своей статье слово «тип» (type), и целые поколения программистов не так поняли, что он имел в виду.

Если вы читали статью Симони внимательно, то знаете, что он предложил соглашение об именах переменных, похожее на использованное мной в приведенном выше примере, где мы решили, что `us` обозначает «опасную строку», а `s` – «безопасную строку». Обе эти переменные имеют тип `string`. Компилятор не помешает вам присвоить одной из них значение

[1] Между прочим, вышло третье издание книги Скотта «Эффективный C++» («Effective C++»), полностью переработанное.

другой, и IntelliSense ничего не подскажет. Но семантически они различаются; они должны интерпретироваться и рассматриваться по-разному, и если вы присваиваете значение переменной одного типа переменной другого типа, то нужно вызывать функцию для конвертации, иначе возникнет ошибка *времени выполнения* (runtime bug). *Если* вам повезет.

Оригинальную концепцию нотации Симони в Microsoft назвали Apps Hungarian («венгерской для приложений»), потому что применялась в отделе приложений (Applications Division), который, в частности, разрабатывал Word и Excel. В исходном тексте Excel вы встретите много rw и col и, видя их, сразу поймете, что они относятся к строкам и столбцам. Да, это целые числа, но бессмысленно присваивать значение строки столбцу, и наоборот. В Word, как мне говорили, есть многочисленные xl и xw, где xl означает «горизонтальная координата относительно листа», а xw – «горизонтальная координата относительно окна». Обе переменные целые. Не взаимозаменяемые. В обоих приложениях часто встречается cb, что означает «счетчик байтов». Да, это тоже целое, но чтобы узнать это, достаточно посмотреть на имя переменной. Это счетчик байтов – размер буфера. И если вы видите xl = cb, то это сигнал: очевидно, что код здесь – неправильный, потому что даже при том, что и xl, и cb являются целыми, полный идиотизм присваивать горизонтальной координате размер буфера в пикселах.

В нотации Apps Hungarian префиксы используются не только для переменных, но и для функций. Честно говоря, я никогда не видел исходного текста Word, но готов держать пари на доллар против цента, что в нем есть функция по имени YlFromYw, преобразующая вертикальные координаты относительно окна в вертикальные координаты относительно листа. Нотация Apps Hungarian требует вместо более привычного TypeToType использовать TypeFromType, чтобы имя каждой функции начиналось с типа данных, которые она возвращает, – точно так же, как это сделал я в своем примере, переименовав функцию Encode в SFromUs. По правилам Apps Hungarian вы обязаны дать функции Encode название SFromUs. Apps Hungarian не оставляет вам выбора в названии функции. И это хорошо, потому что меньше нужно запоминать и не придется задаться вопросом, какую кодировку осуществляет Encode, – у вас есть нечто гораздо более точное.

Нотация Apps Hungarian была чрезвычайно ценна, особенно в эпоху программирования на C, когда компилятор предоставлял не слишком удобную систему типов.

Но затем случилась неприятность.

Силы зла завладели венгерской нотацией.

Никто в точности не знает, но похоже, что это авторы документации из команды Windows неосторожно изобрели то, что стали называть Systems Hungarian (системная венгерская).

Кто-то где-то прочел статью Симони, в которой говорилось о «типе», и решил, что автор имел в виду тип как класс, как в языке с системой типов, как проверку типов, выполняемую компилятором. Автор имел в виду не это. Он тщательно и точно объяснил, что подразумевал под словом «тип», но это не помогло. Вред был нанесен.

В Apps Hungarian были определены очень полезные и содержательные префиксы, такие как `ix` – для обозначения индекса в массиве, `c` – для счетчиков, `d` – для разности между двумя числами (например, префикс `dx` означал «ширину») и так далее.

В Systems Hungarian были гораздо менее полезные префиксы, такие как `l` – для длинного целого (long), `ul` – для беззнакового длинного целого (unsigned long) и `dw` – для двойного слова (double word), которое, фактически, э-э, является беззнаковым длинным целым. В Systems Hungarian префикс сообщал вам лишь одно, а именно: фактический тип данной переменной.

Это было скрытым, но совершенно ошибочным пониманием намерений и практики Симони, и должно служить примером того, что если написать сложный и непонятный научный текст, то никто его не поймет, ваши идеи будут извращены, а затем эти извращенные идеи будут высмеяны, несмотря на то что они никогда не были вашими. Так, в Systems Hungarian появлялись всякие `dwFoo`, имена которых говорили вам, черт возьми, только то, что эта `foo` занимает двойное слово, что, в общем-то, не сообщает вам ничего полезного. Неудивительно, что у Systems Hungarian появились противники.

Нотацию Systems Hungarian распространили; она стала стандартом во всей документации по программированию Windows; ее широко рекламировали такие книги, как «Программирование в Windows» Чарльза Петцольда («Programming Windows», Charles Petzold), настоящая библия для изучающих программирование в Windows, и она быстро стала доминирующим видом венгерской нотации даже в Microsoft, где только очень немногие программисты помимо разработчиков Word и Excel поняли, какая ошибка была сделана.

А затем пришел Великий Бунт. В конечном счете, программисты, которые никогда не понимали венгерскую нотацию правильно, первые заме-

тили, что то неправильно понятое подмножество, которое они использовали, было Раздражающим и Почти Бесполезным, и восстали против него. Однако у Systems Hungarian все-таки есть хорошие качества, помогающие видеть ошибки. По крайней мере, применяя Systems Hungarian, можно сразу узнать тип переменной там, где вы ее используете. Но это несравнимо с достоинствами Apps Hungarian.

Пик Великого Бунта совпал с первым выпуском .NET. Microsoft наконец начала говорить, что пользоваться венгерской нотацией не рекомендуется. По этому поводу было много веселья. По-моему, они даже не потрудились объяснить причин. Просто открыли раздел руководства, посвященный правилам именования, и написали «Не используйте венгерскую нотацию» во всех статьях. К этому времени венгерская нотация была уже так непопулярна, что никто не стал жаловался, и все, кроме разработчиков Excel и Word, только облегченно вздохнули, избавившись от неуклюжего соглашения об именах, в котором, как они считали, не было нужды в эпоху строгой проверки типов и IntelliSense.

Но в нотации Apps Hungarian все-таки очень много полезного, поскольку она увеличивает сорасположение в коде, что облегчает чтение, написание, отладку и сопровождение кода, а также, что важнее всего, она заставляет неправильный код выглядеть неправильно.

Прежде чем двинуться дальше, я должен сделать то, что пообещал, – еще раз пнуть обработку исключений (exceptions). В предыдущий раз у меня были большие неприятности. В замечании, небрежно брошенном на главной странице «Joel on Software», я написал, что не люблю исключения, потому что фактически за ними кроется `goto`, а это, как я рассуждал, хуже явного `goto`. Разумеется, мне в глотку вцепились миллионы. Единственным из всех, кто вступился за меня, был, конечно, Реймонд Чен (Raymond Chen), между прочим, лучший в мире программист – это кое-что значит, не правда ли?

Вот пример с обработкой исключений в контексте данной статьи. Глаз учится замечать неправильные вещи, когда они выделяются, и это предотвращает ошибки. Для того чтобы при просмотре кода выявлять ошибки, делая код действительно надежным, нужно применять соглашения о кодировании, поддерживающие правильное расположение информации. Другими словами, чем больше информации о том, что делает код, расположено прямо перед вашими глазами, тем эффективнее окажется поиск ошибок. Допустим, есть код:

```
dosomething();
cleanup();
```

Замечаете, что здесь что-то не так? «Мы всегда делаем уборку!» Но если dosomething вызовет исключение, то до cleanup дело может не дойти. Это легко исправить с помощью finally или чего-то подобного, но я не об этом; я о том, что единственный способ узнать, что cleanup точно вызовется, это исследовать все дерево вызовов dosomething и разобраться, нет ли где-нибудь чего-нибудь такого, что может вызвать исключения, и это правильно, и есть такие вещи, как проверяемые исключения, которые облегчают жизнь, но я хочу указать на то, что исключения мешают близкому расположению информации. Чтобы узнать, правильно ли работает код, вы должны обращаться в какое-то другое место и потому не можете воспользоваться естественной способностью вашего глаза видеть неправильный код, потому что смотреть не на что.

Когда я пишу маленький скрипт, чтобы раз в день собрать какие-то данные и напечатать их, то да, исключения работают великолепно. Нет ничего приятнее, чем забыть про возможные неприятности и просто обернуть всю программу в один большой обработчик исключений try/catch, который пошлет мне по электронной почте сообщение, если что-нибудь случится. Исключения прекрасно подходят для кода «на скорую руку» – скриптов и кода, от которого не зависит ничья жизнь и судьба. Но если вы пишете операционную систему, или систему управления атомной электростанцией, или приложение для управления медицинской техникой, применяемой при операции на сердце, исключения чрезвычайно опасны.

Знаю, некоторые решат (и напрасно), что я плохой программист, если не могу разобраться в исключениях и в том, как они могут улучшить мою жизнь, если только я допущу их в свое сердце. Чтобы писать действительно надежный код, нужно применять простые инструменты, которые учитывают свойственную человеку ненадежность, а не сложные инструменты со скрытыми побочными эффектами и дырявыми абстракциями, предполагающими, что программист не делает ошибок.

Дополнительная литература

Если вы все еще остаетесь фанатиком исключений, прочитайте статьи Раймонда Чена «Cleaner, more elegant, and harder to recognize» («Чище, элегантней и сложней для понимания») и «Cleaner, more elegant, and wrong» («Чище, элегантней и неправильно») (*blogs.msdn.com/oldnewthing/*

archive/2005/01/14/352949.aspx): «Крайне тяжело увидеть различие между плохим кодом, основанным на исключениях, и не-плохим кодом, основанным на исключениях... исключения слишком сложны, и я не достаточно умен, чтобы с ними работать».

В рассуждениях о Смерти от Макросов, «A Rant Against Flow Control Macros» («Речь против макросов, управляющих исполнением программы», *blogs.msdn.com/oldnewthing/archive/2005/01/06/347666.aspx*), Раймонд описывает другой случай, когда невозможность получить всю информацию в одном и том же месте делает невозможной сопровождение кода. «Когда вы видите код, который использует [макросы], вам приходится копаться в заголовочных файлах, чтобы понять, как это работает».

Для изучения венгерской нотации начните с оригинальной статьи Симони «Hungarian Notation» («Венгерская Нотация», *msdn.microsoft.com/en-us/library/aa260976(VS.60).aspx*). Дуг Кландер (Doug Klunder) объяснил ее команде Excel в несколько более понятной статье «Hungarian Naming Conventions» («Венгерские соглашения об именовании», *www.byteshift.de/msg/hungarian-notation-doug-klunder*). Чтобы узнать больше о венгерской нотации и о том, как ее обрушили составители документации, прочтите пост Ларри Остермана (Larry Osterman) (*blogs.msdn.com/larryosterman/archive/2004/06/22/162629.aspxf*), в особенности комментарий Скотта Людвига (Scott Ludwig) (*blogs.msdn.com/larryosterman/archive/2004/06/22/162629.aspx#163721*) или сообщение Рика Шота (Rick Schaut) (*blogs.msdn.com/rick_schaut/archive/2004/02/14/73108.aspx*).

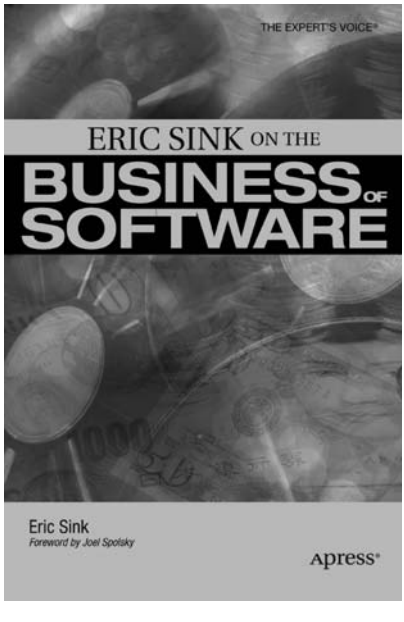
ЧАСТЬ ШЕСТАЯ

Как начать свой бизнес в программировании

ГЛАВА ДВАДЦАТЬ ЧЕТВЕРТАЯ

Предисловие к книге Эрика Синка «Бизнес для программистов. Как начать свое дело»

7 апреля 2006 года, пятница



Эрик Синк (Eric Sink) выступает на форуме «Joel on Software» с его первых дней. Он был одним из создателей веб-броузера Spyglass, создал текстовый процессор AbiWord с открытым кодом, а сейчас занимается разработками в компании SourceGear, производящей программы для управления исходным кодом.

Но большинству из нас он известен своими выступлениями в дискуссионной группе «The Business of Software» (софтверный бизнес), ставшей центром общения всех, кто интересуется софтверными стартапами. Он придумал термин «микро-ISV», в течении нескольких лет пишет о программном бизнесе в своем блоге и написал ряд важных статей для MSDN. Только что он опубликовал полновесную бумажную книгу «Eric Sink on the Business of Software»[1] (Apress, 2006) и попросил меня написать для нее предисловие, которое я вам здесь и предлагаю.

Я когда-нибудь рассказывал о том, как впервые открыл собственное дело?

Постараюсь припомнить все подробности. Мне было лет четырнадцать. В Университете Нью-Мексико было организовано нечто вроде летнего института TESOL (ассоциации преподавателей английского языка), и я устроился туда на работу, которая состояла в том, что я должен был сидеть за конторкой и делать копии статей из журналов для всех, кому это было нужно.

Рядом со мной стоял большой электрический кофейник, и все желающие сами наливали себе кофе, оставляя в стаканчике четверть доллара. Сам я не пил кофе, но любил пончики и подумал, что вкусные пончики должны хорошо пойти с кофе.

Рядом не было магазинов, торгующих пончиками, а так как я был еще слишком мал, чтобы водить автомобиль, то был полностью отрезан от пончиков из Альбукерка. Однако я договорился с одним студентом, чтобы он каждый день покупал пару дюжин пончиков и привозил их мне. Написав от руки объявление «Пончики: 25 центов (Дешево!)», я наблюдал, как потекли деньги.

Каждый, кто проходил мимо меня, увидев объявление, бросал денежку в стаканчик и брал пончик. Появились постоянные клиенты. Ежедневное потребление пончиков непрерывно росло. Даже те, кому ничего не нужно было в институте, отклонялись от своего ежедневного маршрута, чтобы купить у нас пончики.

Разумеется, у меня было право бесплатной пробы, но доходы предприятия от этого не страдали. Стоили пончики, наверное доллар за дюжину. Некоторые покупали пончик за доллар, потому что им было лень искать в стаканчике сдачу. Я не верил своим глазам!

---

[1] Эрик Синк «Бизнес для программистов. Как начать свое дело». – Пер. с англ. – СПб.: Питер, 2008.

К концу лета я продавал по два больших подноса в день – наверное, с сотню пончиков. Накопилась довольно приличная сумма. Точно не помню какая, но порядка нескольких сотен долларов. Надо сказать, это было в 1979 году, и тогда на эти деньги можно было скупить, наверное, все пончики на свете, но к тому времени я уже видеть их не мог, полюбив очень ароматные блинчики с сыром.

Что же я сделал с этими деньгами? Да ничего. Их все забрал глава факультета лингвистики. Он решил организовать на эти деньги большую вечеринку для сотрудников института. Меня на нее не пригласили, потому что я был слишком юн.

Есть ли здесь мораль?

Да никакой.

Но, наблюдая за тем, как растет новый бизнес, испытываешь удивительный восторг. Это радость – видеть стадию органического роста, через которую проходит каждый здоровый бизнес. Под «органическим» я разумею буквально «состоящий из углерода или его соединений». Нет, погодите, я не то хотел сказать. Я имел в виду плавный рост, как у растения. На прошлой неделе ты заработал 24 доллара. На этой – 26. На следующий год в то же время будет, наверное, 100.

Людям нравится развивать бизнес по той же причине, что нравится работать в саду. Как это здорово – посадить в землю семечко, каждый день поливать его, удалять сорняки и смотреть, как из крошечного побега вырастает огромный куст роскошных хризантем (если повезет) или жгучей крапивы (если вы ошиблись во время посадки – но не горюйте, крапиву можно заваривать, только не берите ее голыми руками).

Глядя на доходы, которые приносит ваш бизнес, вы говорите: «Черт, еще только три часа, а у нас уже девять покупателей! Это будет наш самый удачный день!» А через год девять покупателей покажутся вам шуткой, а через два вы обнаружите, что полученный через интранет отчет о продажах за прошлую неделю настолько велик, что с ним невозможно работать.

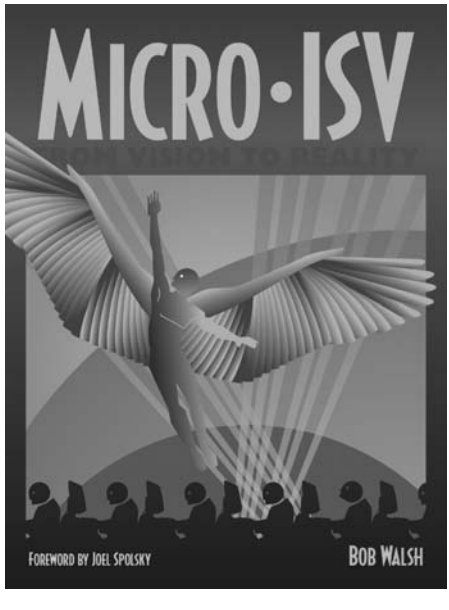
В один прекрасный день вы отключите функцию, которая посылает вам электронное письмо каждый раз, когда кто-то покупает ваш продукт. Это важная веха.

В конце концов, вы обнаружите, что один из ваших практикантов, принятых на лето, притаскивает утром в пятницу пончики и продает их по доллару. И мне хочется верить, что вы не отберете у него заработанные деньги и не устроите на них вечеринку, куда он не приглашен.

ГЛАВА ДВАДЦАТЬ ПЯТАЯ

Предисловие к книге «Micro-ISV: от мечты к реальности»

11 января 2006 года, среда



Это мое предисловие к новой книге Боба Уолша (Bob Walsh) «Micro-ISV: From Vision to Reality» (Micro-ISV: от мечты к реальности), Apress, 2006 .

Как *меня* угораздило таскать рекламные щиты в пользу движения микро-ISV?

Это меня-то!

Когда я организовывал Fog Creek Software, никаких «микро» у меня и в мыслях не было. Я собирался создать крупную транснациональную софтверную компанию с отделениями в 120 странах и штаб-квартирой в небоскребе на Манхэттене, с вертолетной площадкой на крыше, чтобы быстрее добираться до Хэмптонса. Пусть на это потребовались бы десятилетия – мы разворачивались с нуля и всегда планировали медленное и осторожное развитие, – но наши желания никогда не были скромными.

И сам этот термин микро-ISV мне не нравится. «ISV» означает «независимый производитель программного обеспечения» (independent software vendor). Это искусственное слово придумано Microsoft и должно обозначать «компанию-производитель программ, не входящую в Microsoft», точнее, «софтверную компанию, которую мы по каким-то причинам пока не купили и не удалили с рынка, может быть потому, что она занимается чем-то милым, вроде свадебных украшений, слишком экзотическим, чтобы мы до этого снизошли, так что пускай эти человечки радуются жизни. И не забывают использовать .NET!»

Есть еще один термин, «legacy» (унаследованный, морально устаревший), который Microsoft применяет ко всем программам, сделанным не в Microsoft. Поэтому, называя Google «legacy search engine», они намекают, что Google – всего лишь «убогая старая поисковая система, которой вы пользуетесь в силу исторических обстоятельств, пока не смиритесь с неизбежным и не перейдете на MSN». Примерно *так*.

Я предпочитаю говорить «софтверная компания», а в том, чтобы быть начинающим, нет ничего дурного. Называя себя «начинающей софтверной компанией», мы не видим необходимости определять свои отношения с Microsoft.

Полагаю, вы выбрали эту книгу (и правильно сделали), поскольку хотите создать небольшую софтверную компанию, так позвольте мне с этой трибуны огласить свой список из трех пунктов, необходимых для организации микро... тьфу, начинающей софтверной компании. Все остальное, что вам понадобится, описано в книге Боба, но сначала мой вклад.

Первое. Не затевайте бизнес, если не можете объяснить, кого и от какой проблемы вы собираетесь избавить, каким образом ваш продукт решает эту проблему и сколько клиенты готовы заплатить вам за решение их проблемы. На днях я был на презентации шести высокотехнологичных стартапов. *Ни в одной* из этих фирм не было четкого представления о том, какую проблему они решают. Один стартап разрабатывал средство, позволяющее друзьям назначить встречу в кафе, другой – плагин для броузера,

отслеживающий все ваши перемещения в Сети с возможностью удаления ненужных адресов, третий – возможность оставлять текстовые сообщения, привязанные к определенному месту (так что ваш приятель мог получить сообщение, оставленное вами в баре, откуда вы ушли раньше, чем он явился). Сходство между всеми этими компаниями в том, что ни одна из них не решает насущную проблему, и все они обречены, как длиннохвостый кот в комнате, полной кресел-качалок.

Второе. Не затевайте бизнес в одиночку. Да, многие из стартапов, организованных единственным человеком, достигли успеха, но еще больше провалилось. Если вы не убедили в перспективности своей идеи хотя бы одного приятеля, может быть, она... э-э... не столь хороша? Кроме того, в одиночестве работать скучно, и не с кем обсудить возникшую мысль. А когда станет невмоготу – что неизбежно для единственного работника, – вы просто закроете свою лавочку. Будь у вас напарник, вы чувствовали бы перед ним свою обязанность выстоять. P. S. Кошки не в счет.

Третье. Не ждите, что сразу достигнете успеха. Невозможно предугадать, сколько вы заработаете за первый месяц продаж. Начав пять лет назад продавать FogBugz, мы не имели не малейшего понятия, сколько выручим в первый месяц, 50 000 долларов или ничего. Оба варианта были для меня равновероятны. Беседуя с многими предпринимателями, я собрал достаточно данных, чтобы дать вам точный ответ насчет *вашего* стартапа.

Да-да, мой магический кристалл поможет предсказать то, что вам очень хочется узнать, – точную сумму дохода в первый месяц после выпуска вашего продукта.

Готовы?

OK.

В первый месяц вы должны заработать

примерно

364 долллара, *если все сделаете правильно*. Занизите цену – получите 40 долларов, завысите – ничего. Если вы рассчитывали на большее, то разочаруетесь, все бросите, наймете на работу к Бизнесмену и станете называть всех нас, трудящихся в стартапах, «морально устаревшими микро-ISV».

364 доллара... звучит тоскливо, но не пугайтесь, потому что вскоре вы обнаружите ту роковую ошибку, которая мешает половине ваших возможных клиентов раскошелиться, и станете зарабатывать 728 долларов в месяц. И тогда вы будете стараться изо всех сил, приобретете хоть неболь-

шую, но известность, научитесь эффективно применять AdWords, а рассказ о вашей компании появится в местном бюллетене свадебных объявлений, и тут вы уже начнете зарабатывать 1456 долларов месяц. После выпуска версии 2.0 со спам-фильтром и встроенным интерпретатором Common Lisp ваши клиенты начнут общаться между собой, и вы будете делать 2912 долларов в месяц. Затем вы немного играете с ценами, предоставляте платную поддержку, выпускаете версию 3.0, а Джон Стьюарт упоминает ваше название в ежедневном телешоу – и выручка доходит до 5824 долларов в месяц.

Процесс пошел с огоньком. Осталось покорпеть несколько лет, удваивая доходы каждые год-полтора, и независимо от скромного начального уровня [точная математическая формула удалена. – *Ред.*] вы скоро начнете строить на Манхэттене небоскреб с вертолетной площадкой, чтобы добираться до 20-акрового поместья в Саутгемптоне ровно за полчаса.

Мне кажется, в этом и заключается истинная радость организации новой компании: создать собственное детище, лелеять его, вкладывая все свои силы, видеть, как оно растет, как окупаются ваши капиталовложения. Чертовски увлекательное занятие, и я не променяю его ни на что на свете.

ГЛАВА ДВАДЦАТЬ ШЕСТАЯ

Взять высокую ноту

25 ИЮЛЯ 2005 ГОДА, ПОНЕДЕЛЬНИК

В марте 2000 года я запустил сайт «Joel on Software», сделав при этом сомнительное утверждение «большинство людей ошибается, полагая, что для создания успешной софтверной компании нужна уникальная идея» (*www.joelonsoftware.com/articles/fog0000000074.html*):

> *Считается, что для создания софтверной компании главное – найти оригинальную идею решения некой проблемы, с которой до того не могли справиться, реализовать эту идею и в результате сколотить себе состояние. Мы бы назвали это мифом об изобретении наилучшей мышеловки. На самом деле, задача софтверной компании – превращать капитал в работающие программы.*

Последние пять лет я проверяю эту теорию на практике. Формула компании, которую мы с Майклом Прайором (Michael Pryor) создали в сентябре 2000 года, включает четыре элемента:

Это довольно удобная формула. Особенно с учетом того, что нашей *истинной* целью при создании Fog Creek было создать такую компанию, в которой *мы и сами хотели бы работать*. Тогда я утверждал, что лучшие

условия труда (или «создание такой софтверной компании, где захотят работать лучшие в мире программисты») *ведут* к прибыли так же естественно, как шоколад – к лишнему весу, а мультяшный секс в видеоиграх – к гангстерской стрельбе.

Сегодня я хочу обсудить только один вопрос, потому что если ответ на него не верен, то и вся моя теория распадается. Так вот, вопрос: есть ли вообще смысл сотрудничать с «лучшими программистами»? Неужели разница между программистами столь велика, что имеет значение?

Для нас с вами ответ может быть очевиден, однако многим другим требуется доказательство.

Несколько лет назад некая крупная компания хотела приобрести Fog Creek. Я знал, что этому не бывать, поскольку слышал, что директор этой компании заявил, что не разделяет мои взгляды насчет работы с лучшими специалистами, ссылаясь на библию: мол, нужен один царь Давид и армия солдат, умеющих выполнять приказы. Акции его компании упали с 20 до 5 долларов, и хорошо, что мы тогда отказались от его предложения, правда, вряд ли во всем виноват культ царя Давида.

На самом деле, в мире журналистов, пишущих о бизнесе под копирку, и руководителей крупных фирм, которым все разжевывают сверхдорогие консультанты по управлению, принято считать, что главное – уменьшить *стоимость* программистов.

Есть отрасли, где дешевизна *действительно* важнее качества. Компания «Уолмарт» стала крупнейшей в мире корпорацией, продавая дешевые, а не высококачественные товары. В противном случае цены в ее магазинах выросли бы, и все конкурентное преимущество, состоящее в дешевизне, исчезло. Если бы они попробовали продавать, скажем, носки, выдерживающие издевательства вроде стирки в стиральной машине, им пришлось бы использовать для их производства дорогие материалы, например *хлопок*. И тогда бы все носки стали дороже.

Так почему же в софтверной области нет места поставщику дешевых услуг – компании, использующей труд самых низкооплачиваемых программистов? (Не забыть спросить в Quark, как все-таки работает эта концепция уволь-всех-и-найми-самых-дешевых.)

Дело вот в чем: тиражирование программного обеспечения не стоит ничего. Значит, стоимость программистов делится на все проданные вами экземпляры программы. В области программного обеспечения можно

улучшить качество продукта, не увеличивая затраты на каждую проданную единицу.

По существу, *разработка программного продукта увеличивает его ценность быстрее, чем затраты на его производство.*

Грубо говоря, урезав расходы на программистов, вы выпустите дрянной продукт, но при этом мало сэкономите.

Примерно то же самое происходит в индустрии развлечений. Есть смысл пригласить Брэда Питта сняться в вашем блокбастере, несмотря на то что он затребует весьма высокий гонорар, потому что его гонорар можно разделить на миллионы зрителей, которые пойдут на этот фильм только потому, что Брэд так *хорош*.

Или, другими словами, есть смысл пригласить Анджелину Джоли сняться в вашем блокбастере, несмотря на то что она затребует весьма высокий гонорар, потому что ее гонорар можно разделить на миллионы зрителей, которые пойдут на этот фильм только потому, что Анджелина так *хороша*.

Однако я все еще хожу вокруг да около. Что значит «быть лучшим программистом», и так ли разнится софт, написанный разными программистами?

Начнем со старого вопроса продуктивности. Измерить продуктивность работы программиста достаточно сложно. Практически все предлагаемые метрики (строки отлаженного кода, количество методов, количество аргументов командной строки) легко обмануть, а получить конкретные данные по большим проектам очень трудно, потому что крайне редко два программиста получают одинаковое задание.

Я основываюсь на данных профессора Стэнли Айзенштата (Stanley Eisenstat) из Йеля. Он ежегодно читает интенсивный курс CS 323, где большая часть практической работы состоит примерно из пяти заданий по программированию, рассчитанных на две недели каждое. Задачи достаточно сложны для обычного курса: разработать оболочку командной строки для UNIX, программу ZLW-сжатия файлов и так далее.

Студенты так горячо спорили по поводу объема работ, выполняемых на этом курсе, что профессор попросил студентов сообщать ему, сколько времени у них заняла каждая задача. Он собирал эти данные в течение нескольких лет.

Я потратил некоторое время на изучение этих данных. Это единственные известные мне статистические сведения, собранные по результатам

работы десятков студентов, получивших одинаковые задания и использующих одинаковые технологии в одно и то же время. И они систематизированы.

Первое, что я сделал с этими данными, это вычислил среднее, минимальное и максимальное количество часов, затраченных на каждую из двенадцати задач, а также стандартное отклонение. Вот результаты:

Проект	Ср.	Мин.	Макс.	Ст. откл.
	часов			
CMDLINE99	14,84	4,67	29,25	5,82
COMPRESS00	33,83	11,58	77,00	14,51
COMPRESS01	25,78	10,00	48,00	9,96
COMPRESS99	27,47	6,67	69,50	13,62
LEXHIST01	17,39	5,50	39,25	7,39
MAKE01	22,03	8,25	51,50	8,91
MAKE99	22,12	6,77	52,75	10,72
SHELL00	22,98	10,00	38,68	7,17
SHELL01	17,95	6,00	45,00	7,66
SHELL99	20,38	4,50	41,77	7,03
TAR00	12,39	4,00	69,00	10,57
TEX00	21,22	6,00	75,00	12,11
ВСЕ ПРОЕКТЫ	21,44	4,00	77,00	11,16

Первое, что бросается в глаза, – огромный разброс данных. Самые проворные студенты заканчивали работу в три-четыре раза быстрее средних и – ни много ни мало – в десять раз быстрее самых медленных. Стандартное отклонение огромно. Тогда я подумал: хмм, наверное, некоторые из этих студентов сделали просто-таки отвратительные реализации. И решил исключить студентов, у которых за 4 часа не получилась работающая программа. Поэтому я уменьшил выборку и взял данные только по 25% студентов с лучшим кодом и высшими оценками. Замечу, что оценки в классе профессора Айзенштата *полностью* объективны: они вычисляются по формуле, исходя из количества автоматизированных тестов, пройденных кодом, и ничего больше. За плохой стиль и задержку баллы не снимаются.

Итак, вот результаты лучшей четверти студентов:

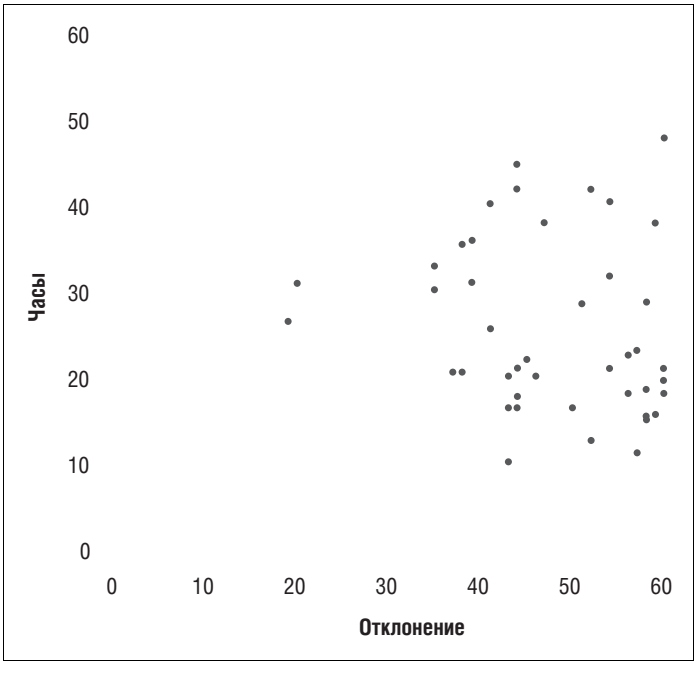
Проект	Ср.	Мин.	Макс.	Ст. откл.
	часов			
CMDLINE99	13,89	8,68	29,25	6,55
COMPRESS00	37,40	23,25	77,00	16,14
COMPRESS01	23,76	15,00	48,00	11,14
COMPRESS99	20,95	6,67	39,17	9,70
LEXHIST01	14,32	7,75	22,00	4,39
MAKE01	22,02	14,50	36,00	6,87
MAKE99	22,54	8,00	50,75	14,80
SHELL00	23,13	18,00	30,50	4,27
SHELL01	16,20	6,00	34,00	8,67
SHELL99	20,98	13,15	32,00	5,77
TAR00	11,96	6,35	18,00	4,09
TEX00	16,58	6,92	30,50	7,32
ВСЕ ПРОЕКТЫ	20,49	6,00	77,00	10,93

Невелика разница! Для четверти студентов (лучших!) практически то же стандартное отклонение. Более того, если внимательно посмотреть на эти данные, станет ясно, что *видимой зависимости между временем и оценкой нет*. Вот типичный график данных для одной из задач. Я выбрал задание COMPRESS01 (реализация алгоритма сжатия Зива-Лемпеля-Уэлша) за 2001 год, потому что стандартное отклонение для него очень близко к общему стандартному отклонению.

Нечего здесь искать, в этом-то вся и суть. *Качество работы и время, затраченное на нее, просто не коррелируют.*

Я спросил об этом профессора Айзенштата, и он обратил мое внимание еще на одну деталь. Поскольку работы следует сдавать к определенному сроку (обычно это полночь), а опоздание влечет существенный штраф, многие студенты прекращают работу, не доведя проект до конца. Иными словами, максимальное время, затраченное на задачу, может оказаться низким потому, что времени между получением задания и сроком, когда его нужно сдать, просто не хватает. Если бы у студентов было неограниченное время для разработки программ (что чуть больше походило бы на реальную жизнь), разрыв оказался бы *еще больше*.

Эти данные нельзя назвать строго научными. Наверняка здесь не все достоверно. Некоторые студенты могли завышать время своей работы



в надежде возбудить к себе сочувствие и получить более легкие задачи в следующий раз. (Как же! Задания CS 323 не менялись с 1980-х, когда этот курс слушал я.) Другие студенты могли занижать затраты, потому что плохо следили за временем. Тем не менее, думаю, не будет большой натяжкой считать, что эти данные показывают пятикратную или десятикратную разницу в продуктивности разных программистов.

Погодите, еще не все!

Если бы единственной разницей между программистами была продуктивность, можно было бы попробовать заменить одного действительно хорошего программиста пятью средними. Но ничего не выйдет. А все потому, что закон Брукса гласит: «Подключение новых людей к опаздывающему проекту задерживает его еще больше» (Frederick Brooks, «The Mythical Man-Month. Essays on Software Engineering»[1], 1995). Один хороший программист,

[1] Фредерик Брукс «Мифический человеко-месяц, или Как создаются программные системы». – Пер. с англ. – СПб.: Символ-Плюс, 2000.

работающий над единственной задачей, не тратит времени на координацию действий и общение с коллегами. Пять программистов, работающих над той же задачей, должны согласовывать свои действия и общаться между собой. Это отнимает массу времени. Есть и другие преимущества работы минимальной командой. Человеко-месяц – это действительно миф.

И это еще не все!

Главный недостаток привлечения множества посредственных программистов вместо немногих первоклассных состоит в том, что сколько бы они ни старались, они никогда не создадут такой продукт, какой могут создать выдающиеся программисты.

Пять Антонио Сальери не напишут «Реквием» Моцарта. Никогда. Даже если будут работать над ним сто лет.

Пять Джимов Дэвисов (это создатель несмешного комикса с котом, где 20% шуток касаются того, какой плохой день понедельник, а остальные посвящены любви этого кота к лазанье – те еще *шуточки*!) за всю жизнь не смогут сочинить ничего подобного эпизоду «Суп Наци» сериала «Сайнфельд».

Команда разработчиков плеера Creative Zen может потратить годы на улучшение своего жалкого подобия iPod и все же никогда не выпустить ничего столь же прекрасного и элегантного, как Apple iPod. И им ничего не удастся откусить от рыночного пирога Apple по той причине, что у них отсутствует чудесный талант дизайнера. *Просто нет его там, и все.*

Просто посредственность *никогда не возьмет высокую ноту,* которую легко берет талант. Мало кто из оперных див может взять фа третьей октавы в партии Царицы ночи Моцарта... Но без этого знаменитого фа исполнить Царицу ночи просто нельзя.

Можно ли сравнивать программное обеспечение с оперным искусством? «Может, у кого-то и так, – возразите вы, – но я-то работаю над интерфейсом пользователя для счетов к получению в индустрии утилизации медицинских отходов». Справедливо. Я имею в виду главным образом компании, разрабатывающие коммерческий продукт, успех или провал которого зависит от его качества. Требования к программам для внутреннего употребления и поддержки своего бизнеса могут быть существенно ниже.

И в последние несколько лет мы видели множество прекрасных программ – настоящих высоких нот, – каких посредственным разработчикам просто *не видать.*

В 2003 году Nullsoft выпустила новую версию плеера Winamp, разместив на своем сайте такое объявление:

- Новый крутой дизайн!
- Новые клевые функции!
- Почти все работает!

Всех развеселил последний пункт – «Почти все работает!». И все счастливы, довольны плеером, пользуются им, рассказывают о нем друзьям, считают, что Winamp – просто фантастика, и все потому, что они взяли и написали на своем сайте: «Почти все работает!». Круто, да?

Если подбросить еще программистов в команду разработчиков Windows Media Player – смогли бы они взять такую высокую ноту? Никогда в жизни. Потому что чем больше команда, тем скорее в ней найдется брюзга, который скажет, что написать у себя на сайте «большинство функций действительно работает» – непрофессионализм и ребячество.

Не говоря уже о комментарии «Winamp 3: чуть новее, чем Winamp 2!».

Вот за это мы и любим Winamp.

Но как только тупицы из AOL Time Warner прибрали эту игрушку к рукам, юмор с сайта исчез. Так и видишь их – злобных, завистливых и лицемерных, как Сальери в «Амадеусе», готовых задушить любое проявление творчества, не одобренное старушкой из Миннесоты, ценой отказа от всего, что заставило людей *полюбить* этот продукт.

Или давайте взглянем на iPod. *Батарею заменить нельзя*. То есть когда аккумулятор умирает, все *очень грустно. Покупай новый iPod*. На самом деле, Apple готова заменить аккумулятор, если вы отошлете свой iPod на завод, но стоит это почти 66 долларов. Вот так-то.

Почему нельзя заменить аккумулятор?

Мое предположение: дизайнеры Apple не захотели испортить идеально-гладкую поверхность своего прекрасного и соблазнительного плеера грубой крышкой аккумуляторного отсека с вечно ломающейся защелкой и швами, в которые набивается мусор из кармана, – как в любом дешевом бытовом приборе. iPod – самое совершенное бытовое электронное устройство из всех, когда-либо виденных мной. Он красив. Он *приятен* на ощупь, как гладкий речной камушек. Крышка аккумулятора могла бы полностью уничтожить это ощущение.

Apple приняла решение в пользу *стиля*. На самом деле, в iPod полно таких «стилевых» решений. А стиль – это нечто недоступное ста программистам Microsoft или двумстам промышленным дизайнерам компании с не-

уместным названием Creative, потому что у них нет Джонатана Айва (Jonathan Ive), и Джонатаны Айвы на дороге не валяются.

Что поделать, об iPod я могу говорить бесконечно. А это прекрасное пощелкивающее колесико управления! Apple потратила *дополнительные средства,* встроив динамик в *сам iPod*, чтобы эти щелчки доносились из-под самого колесика управления. Можно было воспроизводить щелчки через наушники, сэкономив на этом сущие *гроши*. Но с таким колесиком управления вы чувствуете, как послушен плеер. Людям нравится управлять. *Люди счастливы, когда они управляют.* Счастливы. Вы счастливы от того, что колесико *щелкает,* плавно и быстро реагируя на команды. Это не прочий карманный электронный хлам, который так долго запускается, что, нажав выключатель, минуту ждешь первых признаков жизни. Управляет ли человек таким устройством? Вряд ли. Где это видано, чтобы мобильный телефон сразу включался после нажатия кнопки!

Стиль.

Счастье.

Эмоциональная притягательность.

Это то, что делает программное обеспечение, фильм и бытовую электронику хитом. И если вы этого не понимаете, то даже решая проблему, ваш продукт не станет главным хитом, благодаря которому *все* сотрудники вашей компании разбогатеют и станут разъезжать на стильных, счастливых, ярких машинах вроде Ferrari Spider F1, и еще останется на что построить ашрам во дворе.

Дело не в «десятикратной продуктивности». А в том, что «среднепродуктивный» разработчик никогда не берет высокие ноты, делающие программный продукт выдающимся.

К сожалению, это не касается немассовой разработки ПО. Внутренний программный продукт редко бывает настолько важен, чтобы нанимать для его разработки звезд. Долли Партон не приглашают петь на свадьбах. Вот почему более удачную карьеру делают программисты реальных софтверных компаний, а не ИТ-сотрудники разных банков.

Рынок ПО сегодня напоминает игру, в которой «победитель забирает все». Кроме Apple, никто не зарабатывает на MP3-плеерах. Никто не зарабатывает на электронных таблицах и текстовых процессорах, кроме Microsoft. Да, я знаю, что они предприняли определенные шаги, чтобы избавиться от конкурентов, тем не менее факт остается фактом: в этой системе победитель забирает все.

Нельзя позволить себе занимать второе место, поставляя «приемлемый» продукт. Он должен быть очень хорошим, настолько хорошим, чтобы его заметили. Тот дар, который приносят по-настоящему талантливые программисты, – ваш единственный шанс быть замеченным. Вот и вся формула:

ЧАСТЬ СЕДЬМАЯ

Управление софтверным бизнесом

ГЛАВА ДВАДЦАТЬ СЕДЬМАЯ

Бионический офис

25 ИЮЛЯ 2005 ГОДА, ПОНЕДЕЛЬНИК

Ну что ж.

Это заняло *гораздо* больше времени, чем ожидалось.

Наконец-то мы переехали в новый офис Fog Creek на 8-й Авеню, 535, спустя ровно десять месяцев после того, как я решил подыскать помещение взамен старого дома моей бабушки, где мы работали первые несколько лет, разместившись в спальнях и в саду.

Большинство менеджеров софтверных фирм знают, как выглядит хороший офис, как и то, что у них его нет и не будет. Похоже, проблема устройства офиса никогда не решается правильно, ничего не поделаешь. Компания арендует офис на десять лет, при этом менеджер команды разработчиков – последний, чьим мнением насчет дизайна помещения поинтересуются и кто увидит новые стойла... простите, офисные кабинки, только после переезда.

Черт возьми, уж в моей-то собственной компании я мог бы, черт возьми, сделать, как захочу! Вот и сделал.

Возможно, у меня завышенные требования к архитектуре. Вероятно, я чаще среднего программиста замечаю то, что меня окружает. Наверное, я слишком серьезно к этому отношусь. Но виной тому три причины:

- Есть масса свидетельств того, что правильная организация рабочего пространства увеличивает продуктивность программиста; особенно это касается личных кабинетов.

- Предлагая сногсшибательно роскошные личные кабинеты с окнами, гораздо легче нанять на работу суперзвезд, продуктивность которых в десять раз выше, чем у просто блестящих программистов. Если я в Нью-Йорке предлагаю зарплаты на уровне Бангалора, то мне нужны эти суперзвезды, поэтому на собеседовании хочу видеть их с отвалившейся челюстью. И это *драма*.
- В конце концов, это моя *работа*. Здесь я провожу *значительную часть* своей жизни вдали от родных и друзей. Так пусть здесь будет красиво.

В сотрудничестве с архитектором Роем Леоне (Roy Leone), при наличии большой площади (426 квадратных футов на человека) и прогрессивного CEO, я поставил перед собой цель создать совершенное рабочее пространство для разработчиков.

Вот какие «системные требования» (архитекторы называют это иначе) я дал Рою:

1. Личные кабинеты с закрывающимися дверьми абсолютно необходимы, это не обсуждается.
2. Программистам нужно много электрических розеток. Они должны находиться на уровне стола, чтобы можно было подключать разные устройства, не ползая по полу.
3. Необходимо, чтобы любые провода для передачи данных (телефон, сеть, кабельное телевидение, сигнализация и так далее) можно было легко перекладывать, не вскрывая стены.
4. Офис должен позволять программистам работать в паре.
5. Те, кто постоянно сидит за монитором, должны периодически давать отдых глазам, переводя взгляд на удаленные предметы, поэтому мониторы не должны располагаться вдоль стен.
6. Офис должен быть местом для встреч, где можно приятно провести время. Встречаясь с друзьями после работы, чтобы поужинать, вы должны *хотеть* встретиться в офисе. Как поведал Филипп Гринспен (Philip Greenspun) (*ccm.redhat.com/asj/managing-software-engineers/*): «Успех вашего бизнеса зависит от того, насколько ваш офис является для программистов средой обитания. Для этого нужно, чтобы офис был лучше дома среднего программиста. Тут есть два пути. Первый – нанимать программистов из трущоб. Второй – сделать офис красивым».

Рой прекрасно справился с работой. Вот за это мы и платим архитектору. Думаю, Рой может стать чем-то вроде международного эксперта по разработке офисов для программистов. Вот как он реализовал мои системные требования в трехмерном пространстве:

Личные кабинеты. Мы получили не только просторные личные кабинеты с окнами, но и общее помещение с рабочими местами для не-разработчиков, скрытыми в угловых альковах, так что каждый получил свое личное пространство, вне поля зрения других.

Стены между кабинетами и рабочими местами выполнены из высокотехнологичного полупрозрачного акрила, который мягко светится, пропуская естественный свет без ущерба для приватности.

Электронергия. Каждое рабочее место оборудовано двадцатью – именно двадцатью розетками. Четыре из них (оранжевые) подключены к источнику бесперебойного питания в серверной, так что UPS в каждом кабинете не требуется.

Розетки расположены чуть ниже стола в специальном коробе шесть на шесть дюймов, пролегающем вдоль всего стола. В нем можно аккуратно спрятать все кабели и закрыть его удобной крышкой под цвет стола.

Провода. Мы использовали систему Snake Tray, расположенную под потолком. Она начинается в серверном отсеке и проходит вдоль всего офиса, через каждый кабинет. Система полностью доступна, поэтому, если понадобится протянуть любой низковольтный кабель из точки А в точку Б, это делается легко и аккуратно. Мы переехали только в пятницу, и уже переделали офисную локальную сеть, что заняло у нас всего полчаса, так что кабель-канал Snake Tray оправдал возложенные на него надежды. В каждом кабинете есть свой 8-портовый коммутатор, к которому можно подключить свой ноутбук, *и* свой рабочий компьютер, *и* свой Макинтош, *и* тот старый комп, который вы держите у себя для того, чтобы читать «Joel On Software», когда основной компьютер перезагружается, устанавливая свежее обновление Windows, и у вас в запасе остается *еще* 3 порта. (Вниманию математических гениев: не нужно забрасывать меня письмами. Один порт служит для соединения с сервером.) Меня поражают глупые управляющие зданиями, которые все еще считают, что одного сетевого порта на кабинет достаточно. Может, это и так – для адвокатов.

Парное программирование. При установке стандартных Г-образных столов многие разработчики садятся в углу, поэтому, когда им нужно временно поработать с кем-то вместе, заняться программированием в паре или просто показать что-то на экране другому человеку, тому приходится

тянуться через весь стол или смотреть через плечо первого. Чтобы избежать этого, мы заказали длинные и прямые столы, поэтому, где бы ни сидел программист, всегда достаточно места, чтобы другой человек мог поставить свое кресло и сесть рядом.

Отдых для глаз. Хотя мы и установили столы вдоль стен, в стене есть внутреннее окно, в которое видно наружное окно *соседнего* кабинета. Благодаря столь замечательному расположению не нарушается уединение, поскольку, хотя ваше окно и выходит в соседний кабинет, оно направлено так, что из большинства точек кабинета вы видите только маленький краешек другого кабинета и его наружное окно. В результате в каждом кабинете – окна на три стороны, два из них смотрят наружу, что реализует архитектурную схему двухстороннего освещения в каждой комнате. Это большое достижение: попробуйте сами придумать такую планировку, чтобы в обычном здании у каждого работника получился угловой офис. Еще одна причина не жалеть о деньгах, потраченных на хорошего архитектора.

Отдых. Мы оборудовали в офисе небольшую кухню и гостиную с диванами, большой плазменной панелью HDTV и DVD-плеером. Мы также собираемся поставить там бильярд и игровую приставку. Личные кабинеты означают, что вы можете слушать музыку на умеренной громкости без наушников, никому при этом не мешая.

Считайте сами

Месячная аренда нашего офиса при полной занятости составляет примерно 700 долларов на одного работника. Перепланировка уложилась в смету и практически полностью оплачена домовладельцем. Подозреваю, что 700 долларов на человека ближе к верхнему пределу для программистов во всем мире, но если при этом мы можем нанимать людей из 99,9-процентили вместо 99-процентили, дело того стоит.

ГЛАВА ДВАДЦАТЬ ВОСЬМАЯ

Вы в полной...

2 ДЕКАБРЯ 2000 ГОДА, СУББОТА

До вчерашнего дня лицензия нашей компании на программу FogBugz не позволяла ее «реконструировать» (reverse engineering), изучать исходный код или каким-либо образом его изменять. Некоторые пользователи честно спрашивали у нас, сколько стоит лицензия на исходный код, потому что хотели модифицировать некоторые функции.

Хм... А почему собственно лицензия *запрещает* трогать исходный текст? Я не смог найти для этого никаких обоснований. Наоборот, нашлось достаточно контраргументов, чтобы немедленно изменить лицензионное соглашение. Так что теперь сидите и слушайте стариковские воспоминания из моего прошлого опыта.

Давным-давно, в 1995 году я работал в Viacom, где наш маленький отряд отважных первопроходцев делал веб-сайты для разных хозяйств в составе Viacom.

В те дни серверы приложений еще не появились. Sybase была так глупа, что для обращения к ее БД в Интернете вы должны были заплатить 150 долларов за клиентскую лицензию для *каждого пользователя, который подключается к вашему сайту*. Вот-вот должен был выйти веб-сервер Netscape 1.0.

И тут бравая компания Illustra стала всюду рассказывать, что ее СУБД идеально подходит для Сети. Дело в том, что архитектура Illustra позволяла легко добавлять новые типы данных, для чего нужно было написать немного кода на C и скомпоновать его с их СУБД. (Любой программист, работавший с СУБД, почувствует здесь опасность. Код на C? Компонуемый

с СУБД? Ох...) Задумывалось это для поддержки таких экзотических типов данных, как широта/долгота, временные ряды и так далее. А потом возникла Сеть. Illustra написала нечто под названием Web Blade и скомпоновала его со своей БД. Web Blade была сырой системой, которая якобы позволяла извлекать данные из БД и создавать динамические страницы «на лету», что в 1995 году было для всех главной проблемой.

Один мой коллега работал над сайтом электронной торговли, посредством которого Blockbuster мог бы продавать через Интернет – не поверите – компакт-диски. Он решил, что Illustra *превосходно* подойдет для этой задачи. Однако Illustra стоила около 125 000 долларов – вытрясти такую кучу денег из Viacom было непросто и заняло немало времени. Коллега поставил у себя в кабинете бумажный стаканчик с надписью «В фонд Illustra» и таким образом собрал несколько долларов. Главный технолог вел мучительные и долгие переговоры с Illustra. В конце концов сделка была заключена. Мы инсталлировали Illustra и приступили к работе.

К несчастью, произошла катастрофа. Выяснилось, что Web Blade – мягко говоря, сырой и совершенно непригодный для работы продукт. Он падал каждые несколько минут. Когда он все-таки заработал, оказалось, что предлагаемый им единственный язык программирования совершенно уникален – этот язык *не был* полным по Тьюрингу, представляете? Менеджер лицензий постоянно отключал вас, и сайт молча умирал. Построение сайта на Web Blade стало сущим кошмаром для моего коллеги. Так что когда мне сказали, что я буду делать сайт для MTV, я взмолился:

– Только, пожалуйста, можно без Illustra?

– Хорошо, но чем ты это заменишь?

Никаких других серверов приложений в те дни не было. Не было ни PHP, ни AOLServer с TCL, Perl запускал по процессу на каждый запрос, пенициллина у нас тоже не было, и вообще жизнь была ужасна.

Моя репутация была поставлена на карту. И я подумал, что самым страшным в Illustra было то, что она регулярно падала, и с этим ничего нельзя было поделать. Будь у меня исходный код, размышлял я, аварийное завершение можно было бы отловить в отладчике и попытаться найти и устранить его причину. Возможно, придется неделю не спать ночами, отлаживая чужой код, но, по крайней мере, появляется *шанс*. А без исходного кода ты в полной ..., как говорится.

Вот тогда я и получил один из главных уроков в области программной архитектуры: для наиболее ответственных, критических задач нужно ис-

пользовать инструмент с более *низким* уровнем абстракции, чем это было бы в идеале. Например, если вы пишете крутую трехмерную стрелялку (такую как Quake, который вышел примерно тогда же) и хотите обойти конкурентов с помощью *крутейшей* трехмерной графики, не берите готовые графические библиотеки. *Напишите собственную*, потому что это фундамент вашей работы. Те, кто пользуется 3D-библиотеками, скажем, DirectX, делают это потому, что пытаются выделиться чем-то другим, а не качеством графики (может быть, сюжетом игры).

И тогда я решил больше не доверять никаким чужим долбаным серверам приложений и написать свой собственный, на C++, используя низкоуровневый API Netscape Server. Потому что, по крайней мере, тогда я знал бы, что если возникла проблема, то она *в моем коде*, и я в состоянии ее устранить.

И это одно из величайших достоинств программ с открытым кодом (open source) и бесплатных, даже если вы можете себе позволить отвалить 125 000 долларов за какую-нибудь Illustra: по крайней мере, если что-то не будет работать, вы как-то сможете это исправить, и вас не уволят с работы, и все эти замечательные-хотя-и-слишком-активные ребята из MTV не будут на вас коситься.

Собираясь проектировать систему, я должен выбрать инструменты. Хороший проектировщик применяет только те инструменты, которым можно доверять, или те, которые можно починить. «Доверять» не означает, что они произведены какой-нибудь крупной фирмой вроде IBM, которой положено доверять, – это значит, что в глубине души вы чувствуете, что они будут работать, как надо. Например, я знаю, что сегодня большинство Windows-программистов доверяет Visual C++. Они могут не доверять MFC, но MFC поставляется с исходным кодом, так что даже если этой библиотеке нельзя доверять, ее можно *исправить*, обнаружив, как ужасно реализованы асинхронные сокеты. Так что поставить свою карьеру на карту MFC все же можно.

Можно довериться СУБД Oracle, потому что она работает, и это всем известно. Вполне можно рассчитывать и на Berkeley DB, потому что если Berkeley DB напортачит, вы возьмете исходный код и исправите его. Но не стоит рисковать карьерой, используя не слишком известный инструмент с закрытым кодом. Можно экспериментировать, но помните о карьере.

Поэтому я стал думать, как сделать FogBugz беспроигрышной картой для трезвомыслящих инженеров. По счастливой случайности программа поставляется в виде исходного кода – так уж устроены сейчас страницы

ASP. Но это меня не беспокоит. В программе отслеживания ошибок нет никаких чудес и алгоритмов, составляющих коммерческую тайну. Это не бог весть какие высокие технологии. (По правде говоря, в *любой* программе очень мало чудес и алгоритмов, составляющих коммерческую тайну. Тот факт, что довольно легко, дизассемблировав программу, разобраться, как она работает, не столь важен, как полагают специалисты по интеллектуальной собственности.) Меня не беспокоит, что люди читают мой код или модифицируют его для своих целей.

С модификацией исходного кода, который вы купили у производителя, связан другой риск: если производитель выпускает новую версию кода, вы можете потратить уйму времени, перенося в нее свои изменения. Здесь я тоже могу кое-чем помочь: если вы нашли ошибку и послали мне исправление, я включу его в следующую версию. Это делается с целью убедить людей, в том, что: а) FogBugz работает; б) если в программе обнаружится критически важная ошибка, они могут исправить ее и избежать увольнения, и в) если уж так случится, что им придется исправлять ошибки и исправления эти разумны, они попадут в код следующей версии, и жить станет легче.

И тут я уже слышу голоса поборников движения open source: «Осел! Сделай свою программу открытой – и все дела! У открытого кода таких проблем не бывает!» Замечательно. Только вот для существования моей крохотной компании с тремя программистами нужно 40 000 долларов в месяц. Так что мы берем деньги за свои программы, не извиняясь, потому что они того стоят. Мы не объявляем свои программы open source, но, заимствовав там два-три здоровых принципа, гарантируем, что выбор FogBugz становится надежным решением.

ГЛАВА ДВАДЦАТЬ ДЕВЯТАЯ

Простота

9 ДЕКАБРЯ 2006 ГОДА, СУББОТА

Дональд Норман (Donald Norman) считает, что значение простоты сильно преувеличено (*www.jnd.org/dn.mss/simplicity_is_highly.html*): «Но когда настал момент, и журналисты должны были оценить выбранные ими простые продукты, пошли жалобы на отсутствие так называемых „критически важных“ функций. Так чего же хотят люди, требуя простоты? Разумеется, выполнения операции нажатием одной кнопки, но при наличии всех нужных им функций».

Довольно давно я написал такой текст («Joel on Software», Apress, 2004):

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% всех экземпляров программы.

К сожалению, это не одни и те же 20%. Всем нужны разные группы функций. За последние десять лет я слышал о десятках компаний, которые, твердо решившись не учиться на чужом опыте, пытались выпускать «облегченные» текстовые процессоры, реализующие всего 20% функций. Эта история – ровесница PC. Обычно они дают свою программу для отзыва журналисту, и тот пишет отзыв, редактируя его текст прямо в новом текстовом процессоре, а потом журналист пытается найти необходимую ему функцию подсчета слов, потому что у большинства журналистов жесткие

требования по количеству слов, а этой функции нет, так как она попала в «80% никем не используемых», в итоге журналист пишет статью, одновременно пытаясь утверждать, что облегченные программы – это хорошо, а раздутые – плохо, добавляя «не могу пользоваться этой чертовой штукой, потому что она не считает слова».

Короче говоря, 20-процентные продукты очень хороши для начальной раскрутки, потому что их можно изготовить даже с ограниченными ресурсами, чтобы составить контингент пользователей. Это стратегия дзюдо, превращающая слабость в силу, такая как в фильме «The Blair Witch Project», у авторов которого была только любительская видеокамера, но они смогли придумать такой сюжет, что это стало благом. Поэтому вы выдаете «простое» за нечто замечательное, когда по некоторому совпадению ничего большего ваши ресурсы не позволили бы сделать. Всего лишь счастливое совпадение, но получилось действительно замечательно!

Приверженцы простоты напомнят о 37signals и Apple iPod, приведя их в качестве примеров, доказывающих привлекательность простоты для покупателя. Я бы сказал, что в обоих этих случаях успех был обеспечен сочетанием ряда факторов: создание аудитории, евангелизм, ясный и скромный дизайн, эмоциональная притягательность, эстетичность, быстрая реакция, прямая и мгновенная связь с пользователями, соответствие программных моделей моделям пользователей, что повышало юзабилити, и контроль, данный пользователю, – все это *характеристики* одного порядка, в том смысле, что покупатели любят такие качества и готовы заплатить, но ни одно из них нельзя назвать «простотой». Например, одна из *характеристик* iPod – красота, чего у Creative Zen Ultra Nomad Jukebox нет, так что, пожалуйста, заверните мне iPod. В случае iPod красота достигается ясным и простым дизайном, но возможны и другие пути. Hummer эстетически привлекателен именно в силу своей уродливости и сложности.

Думаю, не стоит приписывать успех, скажем, iPod *малочисленности его функций*. Пойдя по такому пути рассуждений, можно придти к выводу, что если убрать из продукта какие-то функции, он станет продаваться еще успешнее. Имея за плечами шестилетний опыт управления собственной софтверной компанией, должен сказать: *ничто* и *никогда* так не увеличивало прибыль Fog Creek, как выпуск новой версии с дополнительными функциями. Ничто другое. Прибавка в итоговой строке нашего баланса от выпуска новых версий с новыми функциями не подлежит никакому со-

мнению. Столь же несомненно, как закон всемирного тяготения. Когда мы пробовали рекламу на Google и различные партнерские схемы или когда в прессе появлялась статья о FogBugz, эффект был едва ощутим. Но при выходе новой версии с новыми функциями мы видим внезапное, явное, существенное и постоянное увеличение дохода.

Если называете «простым» продукт, в котором модель пользователя близко соответствует программной модели, благодаря чему продуктом легко пользоваться, – отлично, флаг в руки и вперед. Если называете «простотой» скромный и ясный внешний вид продукта, имея в виду лишь эстетическое восприятие, например, описав одежду от Ralph Lauren как «саутгемптонский WASP», – отлично, флаг в руки и вперед. Минималистская эстетика сейчас на пике моды. Но если вы думаете, что «простота» означает «не слишком много функций» или «делает что-то одно, но зато хорошо», то при всей вашей честности вы немногого достигнете с продуктом, функции которого намеренно ограничены. Даже в iPod есть «лишний» пасьянс. Даже Ta-da List поддерживает RSS.

Ладно, мне пора... Нужно еще поменять мой сотовый телефон на новый, в котором есть скоростной доступ в Интернет, электронная почта, приемник подкастов и MP3-плеер.

ГЛАВА ТРИДЦАТАЯ

Руби-дуби-ду

23 января 2002 года, среда

Желание переписать весь свой код заново иногда возникает потому, что изначально код разрабатывался не для тех целей, в которых его стали использовать. Он мог быть прототипом, экспериментальным или учебным кодом, демо-версией «на скорую руку» или попыткой за 9 месяцев пройти путь от нуля до Ай-Пи-О. После этого он превратился в нечто большое и бесформенное, ненадежное и не позволяющее развивать его дальше, и все жалуются, старые программисты в отчаянии уходят, а новые не могут разобраться в коде и добиваются от руководства согласия начать все заново, а в это время Microsoft отнимает у них бизнес. Сегодня я хочу вам рассказать, как можно действовать иначе.

FogBugz возникла шесть лет назад в качестве эксперимента, когда я изучал программирование ASP. Довольно скоро она превратилась во внутрифирменную программу отслеживания ошибок. Почти каждый день к ней добавлялись новые полезные функции, пока программа не стала *настолько хороша*, что дальнейшая доработка ей не требовалась.

Многие друзья спрашивали меня, нельзя ли им установить FogBugz в *своих* компаниях. Но в программе было слишком много жестко закодированных вещей, что мешало куда-либо перенести ее с компьютера, на котором она была развернута. Я использовал ряд хранимых процедур SQL Server, поэтому для работы FogBugz требовался SQL Server – слишком дорогой и избыточный продукт для команд из двух человек, желающих воспользоваться нашей программой. И так далее. Поэтому я говорил своим друзьям: «Если вы заплатите мне 5000 долларов за консультационные ус-

луги, я готов потратить пару дней и причесать код так, чтобы он выполнялся с Access вместо SQL Server». Обычно друзьям казалось, что это слишком дорого.

После нескольких таких случаев мне пришло в голову, что я мог бы продать ту же программу троим, взяв с них по 2000 долларов и оставшись с прибылью. Или тридцати покупателям по 200 долларов. Ведь с программами это просто. Поэтому в конце 2000 года Майкл взялся за дело, переработал код так, чтобы он мог выполняться как с Access, так и с SQL Server, вытащил в заголовок все, что касалось конкретного сайта, и мы начали продавать программу. На особый успех я не рассчитывал.

Мне казалось тогда, что есть куча пакетов для отслеживания ошибок, и что каждый программист пишет что-то свое для этой цели, – зачем кому-то покупать наш продукт? Я знал, что начинающие свое дело программисты часто заблуждаются, считая *других* такими же программистами, *как они сами*, с теми же потребностями, и у них возникает нездоровое желание продавать программные инструменты. Отсюда столько мелких компаний, торгующих вразнос безделушками для генерации исходного кода, обнаружения ошибок с отправкой сообщений по почте, расцвечивания синтаксиса в редакторе, FTP и отслеживания ошибок, – всякой ерундой, которая может понравиться только программисту. И я не хотел попасть в *эту* ловушку.

Конечно, все идет не совсем так, как планировалось. FogBugz стала популярной. По-настоящему. Fog Creek обязана ей существенной частью своих доходов, и объем продаж неуклонно растет. Народ все покупает и покупает ее.

Мы выпустили версию 2.0, постаравшись добавить в нее все необходимые функции. Когда Дэвид работал над версией 2.0, мы искренне считали, что на нее не стоит тратить много сил, поэтому он больше заботился о «целесообразности», чем об «элегантности». Некоторые, кхм, *конструктивные проблемы* первоначального решения остались догнивать. Имелись два законченных фрагмента почти *идентичного* кода для отображения главной страницы редактирования данных об ошибке. Операторы SQL были разбросаны по всему HTML-коду, там и сям. Наш скрипучий HTML предназначался для древних браузеров, нашпигованных ошибками и способных рухнуть даже при загрузке *about:blank*.

Да, она работала великолепно, и какое-то время не было ни одной зарегистрированной ошибки. Но внутри код представлял собой «большую помойку». Добавление новых функций было большим геморроем. Чтобы до-

бавить одно поле в центральную таблицу ошибок, требовалось внести полсотни исправлений, и то, что мы забывали исправить, всплывало бы постоянно, вплоть до той поры, когда все заведут собственные бунгало на марсианском взморье, летая на уикенд в собственном ракетоплане.

В другой компании, управляемой каким-нибудь бывшим администратором службы экспресс-доставки, могли бы решить выкинуть весь код на помойку и начать все заново.

Я уже говорил, что не верю в начало с чистого листа? Кажется, я только об этом и говорю.

Как бы то ни было, я решил не начинать все заново, а потратить три недели своей жизни на то, чтобы полностью привести в порядок код. Руби-дуби-ду. Для этого упражнения я установил ряд правил в духе рефакторинга:

1. Не добавлять никакие новые функции, даже самые мелкие.
2. В любой момент времени, при каждом сохранении в системе код должен работать идеально.
3. Я буду выполнять только логические преобразования – то, что делается почти механически и позволяет тут же убедиться, что поведение кода не изменилось.

Я просмотрел все исходные файлы, один за другим, сверху донизу, рассматривая код, обдумывая лучшую структуру для него и внося простые изменения. Вот несколько примеров того, чем я занимался эти три недели:

- Всюду заменил HTML на XHTML. Например, заменил тег `<BR>` на `<br />`, заключил все атрибуты в кавычки, проверил соответствие всех вложенных тегов и валидность всех страниц.
- Удалил все форматирование (теги `<FONT>` и так далее), перенеся его в таблицу CSS.
- Убрал все SQL-инструкции из кода представления и полностью из всей программной логики (того, что маркетологи называют бизнес-правилами). Все это было отправлено в классы, которые я даже не проектировал, – просто не спеша, по мере необходимости добавлял методы. (Кто-то с толстой пачкой карточек 4×6 уже точит карандаш, чтобы выколоть мне глаз. Вы хотите сказать, что не проектировали свои классы?)
- Нашел повторяющиеся блоки кода и создал классы, функции или методы, чтобы убрать повторения. Разбил крупные функции на несколько мелких.

- Удалил оставшийся в основном коде английский текст и поместил его в отдельный файл, чтобы упростить интернационализацию.
- Реструктурировал сайт ASP, сделав одну точку входа вместо кучи разных файлов. В результате то, что раньше давалось с большим трудом, стало стало гораздо проще; например, теперь мы можем выводить сообщения об ошибке ввода в той самой форме, где были введены недопустимые данные, – это *должно* быть просто, если все на своих местах, а я, когда еще только изучал программирование ASP, не размещал все правильно.

Все три недели код заметно улучшался. Для пользователя мало что изменилось. Некоторые шрифты стали лучше благодаря CSS. Я мог прекратить эту работу в любой момент, потому что мой код всегда был на 100% работоспособным (и я на всякий случай сохранял каждую версию на нашем действующем внутреннем сервере FogBugz). У меня практически не было сложных проблем, и я ничего не проектировал, потому что выполнял только простые логические преобразования. Попадались замысловатые куски кода. Обычно это были заплатки, накопившиеся за годы. К счастью, мне удалось сохранить заплатки в целости. Часто я замечал, что, начав все с нуля, снова повторил бы ту же самую ошибку, и она могла бы долго оставаться незамеченной.

Я, в основном, закончил. Как и планировалось, на все ушло три недели. *Изменилась почти каждая строка кода.* Да, я просмотрел каждую строку и изменил почти все. Структура совершенно изменилась. Все функции слежения за ошибками теперь отделены от HTML-функций интерфейса пользователя.

Вот положительные стороны моей работы по приведению кода в порядок:

- Потрачено гораздо меньше времени, чем заняло бы полное переписывание. Допустим (исходя из срока, за который FogBugz пришла в нынешнее состояние), полное переписывание заняло бы год. Таким образом, я сэкономил сорок девять рабочих недель. Эти сорок девять недель представляют *известную* конструкцию кода, которую я сохранил. Мне не приходилось думать, что где-то нужна новая строка. Я просто тупо заменял `<BR>` на `<br />` и двигался дальше. Мне не надо было думать, как организовать загрузку файлов по частям. Она работала. Требовалось только немного причесать ее.

- Я не ввел никаких новых ошибок. Конечно, без пары мелких ошибок, скорее всего, не обошлось. Но я не делал того, что могло бы породить новые ошибки.
- При необходимости я мог в любой момент остановиться и представить работающий продукт.
- График работы был полностью предсказуем. Неделя такой работы – и вы точно знаете, сколько строк обрабатываете за час, что позволяет оценить оставшуюся работу. Вот бы разработчикам Mozilla так, а не вилами по воде.
- Теперь гораздо проще добавить в код новые функции. Эти три недели с большой вероятностью окупятся при реализации следующей крупной новой функции.

Корифеем рефакторинга считается Мартин Фаулер (Martin Fowler), хотя, разумеется, принципы приведения кода в порядок издавна известны программистам. Интересная новая область – инструменты рефакторинга, то есть программы, частично автоматизирующие такую работу. До полного спектра всех необходимых инструментов здесь еще далеко – в большинстве сред программирования нельзя даже изменить имя переменной (с автоматическим изменением всех ссылок на нее). Но ситуация улучшается, и если вы хотите основать небольшую компанию, торгующую вразнос безделушками для автоматизации программирования, или внести полезный вклад в виде open source, перед вами полный простор.

ГЛАВА ТРИДЦАТЬ ПЕРВАЯ

Двенадцать важных советов по бета-тестированию

2 МАРТА 2004 ГОДА, ВТОРНИК

Вот несколько советов по бета-тестированию продукта, рассчитанного на широкую аудиторию, – того, что я называю «коробочным продуктом» (shrink-wrap). Они годятся для любых проектов – как коммерческих, так и open source: главное здесь не то, что вы на этом заработаете – деньги, пристальный интерес или признание коллег, а то, что продукт предназначен для многих пользователей, а *не* для внутреннего употребления.

1. Открытое бета-тестирование неэффективно. Либо тестеров оказывается слишком много (вспомните Netscape), и тогда порядочных данных от них не получишь, либо вы получите слишком мало отчетов от имеющихся тестеров.
2. Лучший способ заставить бета-тестера прислать вам отчет – воззвать к его психологической потребности быть последовательным. Нужно, чтобы он пообещал вам прислать отчет, а еще лучше организовать прием заявлений на участие в бета-тестировании. После таких позитивных действий, как заполнение бланка заявления и установка флажка рядом с текстом «Согласен незамедлительно сообщить свое мнение и прислать отчет об ошибках», гораздо больше людей стремится выполнить свое обещание.
3. Не надейтесь, что сможете пройти полный цикл бета-тестирования меньше, чем за 8-10 недель. Я пытался это сделать, но при всем желании это просто невозможно.

4. Не надейтесь, что сможете выдавать бета-тестерам новую сборку чаще раза в две недели. Я пытался это сделать, но при всем желании это просто невозможно.
5. Не планируйте бета-тестирование, включающее меньше четырех выпусков. Я даже не пытался этого делать ввиду очевидной бесполезности!
6. Если во время бета-тестирования добавить хоть одну мелкую функцию, нужно начать заново отсчет двухнедельного срока и сделать еще три или четыре выпуска. Добавив в CityDesk 2.0 код, сохраняющий пробельные символы, под конец бета-цикла, я совершил одну из своих крупнейших ошибок. В результате в программе появились, мягко говоря, неожиданные побочные эффекты, которые мы могли бы выявить, продлив бета-тестирование.
7. Даже если у вас есть процедура регистрации тестеров, лишь один из пяти зарегистрировавшихся пришлет вам отчет.
8. По нашему правилу, каждому, кто прислал *любой* отчет – не важно, положительный или отрицательный, – полагается бесплатный экземпляр программы. Те, кто ничего нам не прислал, не получат бесплатный экземпляр после бета-тестирования.
9. Минимальное необходимое количество серьезных тестеров (то есть тех, кто пришлет вам трехстраничный отчет о своих впечатлениях) – примерно 100. Если вы кустарь-одиночка, то с большим количеством вам и не справиться. Если у вас есть группа тестеров или менеджеров бета-тестирования, постарайтесь обеспечить по 100 серьезных тестеров каждому, кто сможет заниматься обработкой отчетов.
10. Даже если у вас есть процедура приема заявлений, лишь один из пяти зарегистрировавшихся действительно опробует продукт, прислав вам отчет. Поэтому чтобы получить, например, 300 серьезных тестеров для трех тестеров из отдела контроля, нужно принять 1500 заявлений. Меньше – и данных окажется недостаточно. Больше – и вы утонете в одинаковых отчетах.
11. Обычно бета-тестер работает с программой, как только получит ее, а потом теряет к ней интерес. Ему неинтересно заново тестировать ее каждый раз, когда вы присылаете новую сборку, если только он не начал пользоваться программой ежедневно, что для большинства маловероятно. Поэтому нужно распределять выпуски. Разделите тес-

теров на четыре группы, и каждый новый выпуск раздавайте новой группе, чтобы на каждом этапе было бета-тестирование.

12. Не путайте техническое бета-тестирование с маркетинговым. Я здесь говорил о технических бетах, цель которых – найти ошибки и получить свежие отчеты. Маркетинговые беты предшествуют релизу и предоставляются прессе, крупным клиентам и автору книги «... для полных идиотов», которая должна выйти в один день с продуктом. В маркетинговом бета-тестировании получение отчетов не предполагается (хотя авторы книг склонны к *обильной* ответной реакции по любому поводу, и если не уделить им внимания, все это перекочует в их книги).

ГЛАВА ТРИДЦАТЬ ВТОРАЯ

Семь шагов к замечательной работе с клиентами

19 ФЕВРАЛЯ 2007 ГОДА, ПОНЕДЕЛЬНИК

Как начинающая компания, первые пару лет Fog Creek не могла позволить себе нанять специальных людей для работы с клиентами, поэтому Майкл и я занимались этим сами. На помощь клиентам уходило то время, которое можно было потратить на совершенствование программ, зато мы многому научились, и теперь наша служба по работе с клиентами действует гораздо успешнее.

В итоге мы определили семь условий организации замечательного обслуживания клиентов. *Замечательное* следует понимать буквально: нужно обеспечить такую хорошую поддержку, чтобы клиенты ее *заметили*.

1. Исправляйте все двумя способами

Практически каждая проблема технической поддержки имеет два решения. Поверхностное и скорейшее решение просто устраняет возникшую у клиента проблему. Но после некоторых размышлений обычно находится более глубокое решение – способ предотвратить появление этой проблемы в дальнейшем.

Иногда решение состоит в том, чтобы сделать поведение программы или системы ее установки более интеллектуальным: в нашей нынешней программе установки есть масса проверок особых условий. Иногда достаточно исправить текст сообщения об ошибке. Иногда лучшее, что можно сделать, это написать статью для базы знаний.

Каждый звонок в службу технической поддержки для нас как авиакатастрофа для Национального совета по безопасности на транспорте. При крушении самолета высылают дознавателей, выясняют, что случилось, а затем разрабатывают новую политику, чтобы предотвратить появление данной проблемы в будущем. В авиационной безопасности этот подход работает столь хорошо, что если в США все еще изредка падают авиалайнеры, то вследствие каких-нибудь уникальных, из ряда вон выходящих причин.

Отсюда два вывода.

Первое: очень важно, чтобы у сотрудников техподдержки был доступ к команде разработчиков. Это означает, что техподдержку нельзя передать другой организации: ее нужно разместить в одном здании с разработчиками, чтобы те могли вносить исправления. Многие софтверные компании все еще думают, что сэкономили, перенеся техническую поддержку в Бангалор, или на Филипины, или вообще передав ее другой компании. Да, можно понизить себестоимость каждого случая с 50 долларов до 10, но эти 10 вам придется платить снова и снова.

Высококвалифицированный сотрудник обрабатывает обращение в нашу службу поддержки здесь, в Нью-Йорке, повышая вероятность того, что мы *последний раз* сталкиваемся с таким случаем. Поэтому за 50 долларов мы устраняем целый класс проблем.

Телефонным компаниям, кабельным компаниям и интернет-провайдерам почему-то недоступна эта арифметика. Они передают свою техподдержку самому дешевому провайдеру, какого смогли найти, и платят по 10 долларов снова и снова, исправляя одну и ту же проблему снова и снова, вместо того чтобы исправить ее раз и навсегда в исходном коде. Дешевые центры обработки вызовов не имеют возможности исправить проблему; на самом деле они и *не заинтересованы* в решении проблем, потому что их доход основан на повторении, и им очень нравится снова и снова давать все те же ответы на все те же вопросы.

Второе следствие из двух вариантов исправления заключается в том, что рано или поздно все типичные и простые проблемы оказываются решены, и остаются только очень странные и нетипичные проблемы. Это плюс, потому что их гораздо меньше, и вы экономите массу средств, избавившись от рутинной техподдержки. Минус в том, что вместо шаблонных проблем остается только серьезная отладка. Теперь недостаточно обучить новых сотрудников техподдержки десяти стандартным решениям – вам придется научить их отладке.

Для нас политика «исправления всего двумя способами» полностью окупилась. Мы смогли увеличить продажи *вдесятеро*, увеличив затраты на техподдержку всего вдвое.

2. Предложите сдуть пыль

Раймонд Чен (Raymond Chen) из Microsoft рассказывает о клиенте, который жаловался, что у него не работает клавиатура (*blogs.msdn.com/oldnewthing/archive/2004/03/03/83244.aspx*). Естественно, она не была подключена. Если спросить клиента, подключена ли у него клавиатура, «он чувствует себя оскорбленным и с раздражением заявляет: „Конечно, подключена! Я что, похож на идиота?“, на самом деле ничего не проверяя».

«Вместо этого, – предлагает Чен, – скажите ему: „OK, бывает, разъем немного запылился, и нарушилось соединение. Вы не могли бы вынуть разъем, подуть в него и воткнуть снова?“»

«Тогда он лезет под стол, обнаруживает, что забыл подключить ее (или воткнул не в тот разъем), продувает, подключает и говорит: „Да, это помогло, спасибо“».

Подобным образом можно перефразировать многие просьбы, когда клиент должен что-то проверить. Вместо того чтобы просить его проверить настройку, попросите изменить настройку, а потом сделать как было, «просто чтобы убедиться в том, что программа правильно отображает свои настройки».

3. Превратите клиентов в фанатов

Каждый раз, когда нам для Fog Creek требуются товары с фирменной символикой, мы берем их в Lands' End.

Почему?

Давайте-ка расскажу одну историю. Нам для выставки понадобились футболки. Я позвонил в Lands' End и заказал две дюжины с тем же логотипом, что и на рюкзаках, купленных нами ранее.

Когда футболки прибыли, мы с ужасом обнаружили, что логотип невозможно разобрать.

Оказалось, что рюкзаки были светлее футболок. Цвет ниток, который отлично смотрелся на рюкзаках, оказался слишком темным для футболок.

Я позвонил в Lands' End. Как обычно, человек ответил мне *еще до того, как пошли гудки*. Я почти уверен, что у них есть система, когда следующему

агенту в очереди велят быть наготове, *чтобы клиентам не пришлось ждать ни единого гудка перед тем, как им ответит человек.*

Я изложил свою проблему.

Они ответили: «Не переживайте. Вы можете вернуть их за полную стоимость, и мы переделаем футболки нитками другого цвета».

Я сказал: «До выставки осталось два дня».

Они ответили, что вышлют мне коробку с футболками через FedEx, и я получу ее завтра. А первые футболки могу вернуть, когда мне будет удобно.

Они оплатили доставку в оба конца. Я не потратил ни цента. Хотя им ни к чему была куча футболок с нечитаемым логотипом Fog Creek, они взяли затраты на себя.

Но теперь я рассказываю эту историю всем, кому понадобилось такого рода барахло. Я рассказываю ее каждый раз, когда заходит разговор о системах с телефонным меню. Или об обслуживании клиентов. Их система обслуживания клиентов настолько замечательна, что мне нравится рассказывать о ней.

Когда у клиентов проблема, *и вы ее решаете*, они получают большее удовлетворение, чем если бы проблемы не было совсем.

Все дело в привычных ожиданиях. Для большинства опыт общения с техподдержкой связан с авиалиниями, телефонными и кабельными компаниями, а также с интернет-провайдерами, а все они обычно предоставляют *отвратительную* техподдержку. Такую плохую, что вы и не пытаетесь больше туда звонить, правда? Так что у того, кто позвонил в Fog Creek и немедленно, без голосовой почты или телефонного меню, соединился с учтивым и доброжелательным человеком, который на самом деле *решает проблему*, непременно останется более высокое мнение о нас, чем у тех, кому не случалось с нами пересекаться и кто просто предполагает, что мы такие же, как все.

Разумеется, не стоит нарочно что-то портить ради того, чтобы продемонстрировать нашу превосходную службу поддержки. Многие клиенты просто не позвонят, молча кипя от злости.

Но когда кто-то все же звонит, рассматривайте это как отличную возможность превратить его в *фанатично преданного* клиента, который повсюду вещает о том, как здорово вы работаете.

4. Признавайте свою вину

Однажды утром мне понадобился еще один комплект ключей от квартиры, и перед работой я зашел к слесарю за углом.

13 лет жизни в Нью-Йорке научили меня никогда не доверять слесарям: половина сделанных ими дубликатов не подходит. Поэтому я сходил домой проверить ключи – и точно, один не подошел.

Я вернул его слесарю.

Он сделал новый экземпляр.

Я пошел домой и проверил его.

И *этот* не подошел.

Я уже просто кипел от злости. На полчаса опаздывая на работу, я должен был идти к слесарю *в третий раз*. Я было хотел плюнуть на это дело, но решил дать этому неудачнику еще один шанс.

Я ввалился в мастерскую, готовый дать волю своей ярости.

«Что, опять не подошел? – спросил он. – Давайте, посмотрю».

Он стал разглядывать ключ.

Я был вне себя и прикидывал, как бы получше выразить свой гнев на то, что мне пришлось все утро ходить туда-сюда.

«А! Это я виноват», – сказал он.

И внезапно я совершенно перестал злиться.

Чудо, но слова «это я виноват» полностью обезоружили меня. Их было достаточно.

Он сделал третий дубликат. Я больше не злился. Ключ подошел.

Прожив на этой планете сорок лет, я не мог поверить в то, что всего три слова «это я виноват» полностью изменили мое эмоциональное состояние за мгновенье.

Большинство нью-йоркских слесарей не из тех, кто признает свою вину. Слова «это я виноват» совершенно не в их духе. Но он все же произнес их.

5. Заучивайте непривычные фразы

Раз уж утро все равно пропало, я решил позавтракать в кафе.

Это было одно из классических нью-йоркских кафе, вроде того, что показано в «Сайнфелде». Меню на тридцати страницах и кухня размером с телефонную будку. Это абсурд. Чтобы уместить все на таком маленьком

пространстве, нужна технология из «Стартрека», не иначе. Возможно, они собирают все из атомов прямо на месте.

Я сидел рядом с кассовым аппаратом.

Подошла старушка, чтобы расплатиться. При этом она сказала хозяину:

– Вы знаете, я давно хожу сюда, но сегодня тот официант мне нагрубил.

Хозяин вскипел:

– Да что вы! Он не грубил! Это хороший официант! На него еще никто не жаловался!

Женщина не верила своим ушам. Постоянный клиент, желая помочь хозяину, сообщает, что одного из официантов нужно поучить хорошим манерам, а хозяин спорит!

– Пусть так, но я хожу сюда уже давно, и всегда все были очень вежливы со мной, а тот парень грубит мне, – терпеливо объясняла она.

– Да вы хоть вечность сюда ходите, мне все равно. Мои официанты не грубят, – продолжал орать на нее хозяин. – У меня никогда не было проблем. Зачем вы создаете проблемы?

– Послушайте, если вы будете так со мной обращаться, я сюда больше не приду.

– Да и наплевать! – сказал хозяин. Владея нью-йоркским кафе, вы получаете одно очень большое преимущество: в городе столько людей, что можно обругать любого клиента, и все равно у вас их будет пруд пруди.

– И не возвращайтесь! Не нужны мне такие клиенты!

Ну, орел, подумал я. Тебе за шестьдесят, у тебя кафе – и ты гордишься, что морально победил маленькую старушку. Какой ты мачо! Лучше тебе стало от этой победы? Неужели и впрямь стоило терять постоянного клиента?

Или ты перестал бы чувствовать себя мужчиной, сказав: «извините, я поговорю с ним»?

Когда кто-то жалуется, очень легко поддаться гневу.

Решение заключается в том, чтобы заучить несколько ключевых фраз и *тренироваться произносить их*, чтобы в нужный момент, забыв о своем тестостероне, сделать клиента счастливым.

«Извините, это я виноват».

«Извините, не могу принять эти деньги. Ваш заказ за мой счет».

«Это ужасно! Пожалуйста, расскажите, что случилось, и я сделаю все, чтобы это не повторилось».

Вполне естественно, что сказать: «это я виноват» нелегко. Такова человеческая природа. Но эти три слова могут сделать ваших рассерженных клиентов гораздо счастливее. Так что вам придется их сказать. И сказать так, чтобы вам поверили.

Так что приступайте к тренировкам.

Повторите: «это я виноват» сто раз, стоя утром в душе, пока это не начнет звучать как бессмысленное буквосочетание. Тогда вы сможете сказать это в любой момент.

И еще. Возможно, вы считаете, что признавать свою вину нельзя, иначе вас засудят. Это чушь. Чтобы избежать суда, *не доводите людей до бешенства*. Лучше *решить эту чертову проблему*, признав вину.

6. Упражняйтесь в кукловодстве

Рассерженный хозяин кафе, очевидно, принял все близко к сердцу, в отличие от того слесаря. Когда раздраженный клиент жалуется или изливает свой гнев, хочется защищаться.

В таком споре нельзя выиграть, а если вы будете принимать все на свой счет, станет в миллион раз хуже. Вот тогда вы и слышите, как предприниматель говорит: «Мне не нужны такие клиенты, как ты, засранец!» Они радуются своей пирровой победе. Ну, разве не здорово? Ты, владелец маленького предприятия, можешь отшить своего клиента. Замечательно.

Суть в том, что это плохо как для бизнеса, так и для вашего эмоционального самочувствия. Одержав победу над клиентом, прогнав его, вы не перестанете раздражаться и злиться, а клиент вернет свои деньги через компанию-эмитент кредитных карт и расскажет об этом случае десятку друзей. Как пишет Патрик МакКензи (Patrick McKenzie), «нельзя выиграть, споря со своим клиентом» (*kalzumeus.com/2007/02/16/how-to-deal-with-abusive-customers/*).

Есть только один способ переносить сердитого клиента без лишних эмоций: надо осознать, что он сердится не на вас, а на ваш бизнес, а вы просто подвернулись ему под руку.

И поскольку вы для него – нечто вроде куклы, олицетворяющей ваш бизнес, относитесь и вы к себе, как к кукле.

Представьте себя кукловодом. Клиент вопит на куклу. Он орет не на вас. Он злится на куклу.

Ваша задача выяснить, что должна сказать эта кукла, чтобы сделать из этого человека счастливого клиента.

Вы просто кукловод. Вы не участвуете в споре. Когда клиент говорит, «какого черта вы себя так ведете», он просто играет свою роль (в данном случае, цитирует Тома Смыковски из фильма «Офисное пространство»). Вы тоже играйте свою роль. «Извините. Это моя вина». Выясните, как кукла может сделать клиента счастливым, и прекратите, черт возьми, принимать все так близко к сердцу.

7. Жадность – плохой помощник

Недавно общаясь с теми, кто в этом году обрабатывал большую часть обращений в службу техподдержки Fog Creek, я спросил, какие методы они считают наиболее эффективными при работе с сердитым клиентом.

«По правде говоря, – ответили они, – у нас довольно приятные клиенты. Мы практически не видели разгневанных».

Что ж, хорошо, что у нас приятные клиенты, хотя и довольно странно, что за целый год *не было* гневных звонков. Я-то думал, что работа в центре обработки вызовов *состоит* в ежедневной борьбе с взбешенными клиентами.

«Нет. У нас очень милые клиенты».

Вот что я думаю. Думаю, наши клиенты так милы потому, что им не из-за чего переживать. Им не нужно переживать, потому что наша политика возврата денег либеральна до идиотизма: «Нам не нужны ваши деньги, если вы не испытываете безграничное счастье».

Клиенты знают, что им нечего бояться. Они главные в этом партнерстве. Поэтому они не скандалят.

Одним из лучших решений, когда-либо принятых в Fog Creek, было предоставление 90-дневной гарантии возврата денег без каких-либо вопросов. Попробуйте сами: поработайте с Fog Creek Copilot 24 часа в сутки, а через три месяца позвоните и скажите: «Эй, ребята, мне нужна пятерка на кофе, верните мне деньги за Copilot на день позже срока», – и мы вернем их вам. Попробуйте позвонить на 91-й, или 92-й, или 203-й день. Вы все равно получите их назад. Нам не нужны ваши деньги, если вы не удовлетворены. Я почти уверен, что наша доска объявлений о работе – единственная из существующих, которая вернет вам деньги, если реклама не по-

могла. Это неслыханно, но это означает, что мы получаем очень много заказов на рекламу, потому что клиент ничем не рискует.

За последние шесть лет возврат денег клиентам составил 2% нашей прибыли.

2%.

Кое-что добавлю. Большинство клиентов платят с помощью кредитных карт, и если бы мы не возвращали деньги, кто-то из них обратился бы в свой банк. Это называется возврат платежа. Они получают деньги назад, мы платим за принудительный возврат, и если это случается слишком часто, наши операционные расходы растут.

Вы знаете, сколько у нас принудительных возвратов?

0%.

Я не шучу.

Введя более строгие правила возврата денег, мы добьемся только того, что разозлим нескольких клиентов, которые потом будут ныть и жаловаться в своих блогах. Нам не удержать их денег.

Я знаю софтверные компании, на сайтах которых явно указано, что возврат денег не производится, но если вы позвоните им, они рано или поздно вернут вам деньги, потому что знают, что если не вернут они, то вернет ваш банк. Это наихудший вариант. Вам все равно придется вернуть деньги, а у покупателей не будет теплого смутного ощущения безопасности, поэтому они перед покупкой будут колебаться. Или не купят вовсе.

8. (Бонус!) Дайте сотрудникам техподдержки возможность продвигаться по службе

Последний важный урок, который мы усвоили в Fog Creek, заключается в том, что для работы с клиентами нужны весьма высококвалифицированные сотрудники. Продавец в Fog Creek должен обладать значительным опытом разработки программ и уметь объяснить, почему FogBugz работает так, а не иначе, и почему благодаря этой программе команды разработчиков работают лучше. Специалист техподдержки в Fog Creek не сможет обойтись готовыми ответами на стандартные вопросы, потому что мы исправили программу, избавившись от стандартных вопросов, поэтому специалистам техподдержки приходится *действительно отыскивать и устранять проблемы*, для чего зачастую приходится выполнять отладку.

Многим квалифицированным сотрудникам наскучивает работа в службе техподдержки, и я вполне понимаю их. В порядке компенсации, я не нанимаю людей на эти должности, не предоставив им конкретных перспектив карьерного роста. В Fog Creek техподдержка – всего лишь первый год в трехгодичной программе подготовки руководителей, включающей получение степени магистра управления технологиями в Колумбийском университете. Благодаря этому нам удается привлечь для общения с клиентами и решения их проблем честолюбивых и толковых специалистов, имеющих прекрасные перспективы в карьере. Наконец, мы платим им немного больше, чем в среднем платят на этих должностях (особенно с учетом платы за обучение в размере 25 000 долларов в год), но и отдача от них гораздо ощутимее.

ЧАСТЬ ВОСЬМАЯ

Выпуск программного продукта

ГЛАВА ТРИДЦАТЬ ТРЕТЬЯ

Определение даты поставки

9 апреля 2002, вторник

Одна из самых веских причин составить подробный план состоит в том, что он дает вам основание для сокращения числа функций. Если не получается уложиться в срок, *реализовав* функцию «споем вместе» MP3-чата, предложенную Бобом, можно отказаться от этой функции, не обижая Боба.

Вот мои основные правила для цикла выпуска программного продукта:

1. Установить дату выпуска, которая может быть совершенно произвольной.
2. Составить список функциональных возможностей и отсортировать их по приоритету.
3. При отставании от графика исключать функции с низким приоритетом, так, чтобы уложиться в срок.

Следуя этим правилам, вы вскоре обнаружите, что вас не огорчило исключение функций. Скорее всего, они были чем-то вроде балласта. А если они действительно важны, можно реализовать их в следующей версии программы. Это как редактирование. Если хотите написать блестящую статью из 750 слов, напишите 1500 слов, а затем вычеркните лишние.

Кстати, эти правила помогают не забыть, что функции нужно реализовывать *в порядке приоритета*. Если пустите на самотек, ваши программисты начнут с тех функций, которые им интереснее, и вы не сможете ни выпустить программу вовремя, ни сократить функции, потому что все программисты с головой ушли в какой-то Пересчет Караоке, не сделав даже

меню, и через полгода после предполагаемой даты выпуска у вас есть только этот кошмар – хитроумный прикол и никакой функциональности.

Возникает естественный вопрос: как назначить дату выпуска?

Вполне вероятно, что у вас есть внешние ограничения. Фондовая биржа переходит с обычных дробей на десятичные в такой-то день, и если программа не будет готова к этому сроку, ваша фирма погорит, а вас самого отвезут в доки и пристрелят. Или, к примеру, скоро выйдет очередная версия ядра Linux с *новой* оригинальной системой фильтрации пакетов; ее установят все ваши клиенты, а ваше приложение с ней не работает. Ладно, в таких ситуациях определить дату выпуска легко. Дальше можете не читать. Лучше пойдите на кухню и приготовьте вкусный обед для своих близких.

Все, пока!

Но как выбрать дату выпуска всем нам, остальным?

Есть три возможных подхода.

1. **Частые небольшие релизы.** Это подход «экстремального программирования», наиболее пригодный для проектов с небольшой командой разработчиков и малым количеством клиентов, таких как разработка для внутренних потребностей фирмы.
2. **Каждый год-полтора.** Это типично для коробочных продуктов, настольных приложений и так далее, когда разработчиков в команде больше, а клиентов тысячи или миллионы.
3. **Каждые 3–5 лет.** Это типично для гигантских программных систем и платформ, которые являются целыми мирами сами по себе. В эту категорию попадают операционные системы, .Net, Oracle и, по некоторым причинам, Mozilla. В данном случае количество разработчиков может измеряться тысячами (над одним лишь инсталлятором VS.Net работало 50 человек), и очень велико влияние на поставки других программных продуктов, которые не должны пострадать.

Вот несколько факторов, которые нужно учитывать, принимая решение относительно частоты релизов ПО.

При коротких сроках поставки вы быстро узнаете реакцию клиентов на ваш продукт. Иногда лучше всего работать с клиентом так: показать ему программу, дать возможность поработать и сразу включить высказанные пожелания в очередную сборку, которую вы даете клиенту на следующий же день. Не нужно целый год разрабатывать сложную систему с огромным количеством функций, которыми никто не пользуется, потому что вы будете заниматься только тем, о чем вас просят клиенты. **Когда количество**

клиентов невелико, делайте частые выпуски небольшого объема. Объем каждого выпуска – минимальный кусок кода, выполняющий что-то полезное.

Несколько лет назад мне пришлось разрабатывать систему управления веб-контентом для MTV. Из задания следовало, что система должна использовать базу данных и шаблоны, обеспечивая документооборот, чтобы бесплатные внештатные корреспонденты MTV из колледжей по всей стране могли вводить информацию о клубах, магазинах звукозаписей, радиостанциях и концертах. Я спросил: «А как ваш сайт устроен сейчас?»

«Да мы просто делаем все вручную с помощью BBEdit, – ответили мне. – Конечно, приходится обрабатывать тысячи страниц, но в BBEdit есть прекрасная функция глобального поиска и замены...»

По моим оценкам, можно было разработать всю систему за полгода. «Но я хочу предложить вам другое. Давайте сначала запустим механизм шаблонов. Я смогу сделать его за три месяца, и вы сразу избавитесь от ручной работы. Как только он заработает, мы займемся документооборотом, а вы до поры до времени будете работать через электронную почту».

Они согласились. Идея выглядела замечательно. И знаете, что случилось потом? Как только я запустил для них механизм шаблонов, они поняли, что на самом деле обойдутся и без подсистемы документооборота. Механизм шаблонов оказался полезным и для многих других веб-сайтов, которым тоже не был нужен документооборот. В результате мы отказались от документооборота, сэкономив три месяца, за которые я усовершенствовал механизм шаблонов, оказавшийся более полезным.

Не все клиенты согласятся на роль подопытных кроликов в такой схеме. Те, кто покупает готовые программные продукты, не хотят участвовать в Великом Эксперименте Разработки: им нужно нечто предвосхищающее их потребности. Вместо быстрой реакции разработчиков на запрос новой функциональной возможности пользователь предпочитает получить ее *мгновенно* благодаря тому, что она уже есть в программном продукте, вдумчиво разработанном и прошедшем усиленное юзабилити- и бета-тестирование перед выходом в свет. **Если у вас много платежеспособных пользователей (или вы хотите, чтобы их стало больше), лучше выпускать продукт не слишком часто.**

Если вы выпустите худосочную коммерческую программу просто для того, чтобы выставить что-нибудь на обозрение и «получать отзывы пользователей», то, услышав от них, что «программа мало что умеет», решите, что все в порядке – это же лишь версия 1.0. Но когда через четыре месяца

вы выпустите версию 2.0, все будут думать: «*Та* жалкая программа? Я что, должен заново оценивать ее каждые четыре месяца, чтобы выяснить, стала она лучше или нет?» И еще пять лет кряду люди будут помнить свое первое впечатление от версии 1.0, а переубедить их практически невозможно. Вспомните, что случилось с несчастной Marimba. Они создали компанию с неограниченным венчурным капиталом во времена сверхэнергичной рекламной кампании Java, переманив ключевых разработчиков из команды Java фирмы Sun. Их генеральный директор Ким Полиз (Kim Polese) была *непревзойденным* пиарщиком; когда она продавала Java, Дэнни Хиллис (Danny Hillis) сочинял для нее речи о том, что Java – это *следующий этап развития человечества*; Джордж Гилдер (George Gilder) писал захватывающие статьи о том, как Java полностью перевернет саму *природу* человеческой цивилизации. Монотеизм показался бы *жалкой тенью* по сравнению с той верой в Java, к которой нас призывали. Полиз на *такое* способна. Так что когда вышла Marimba Castanet, ее незаслуженная слава была беспрецедентной, но написана она... всего за четыре месяца. Загрузив ее, все увидели – надо же! – окно со списком, которое скачивало программы. (А чего еще хотеть от четырех месяцев разработки?) Большой облом. Разочарование было таким плотным, что его можно было резать ножом. Да и сейчас, через несколько лет, на вопрос «что такое Castanet?» любой ответит, что это окно со списком, которое скачивает программы. Вряд ли кто-то потрудился еще раз оценить ее, а ведь код Marimba писали еще шесть *лет*; уверен, что теперь это крутейшая программа, – но, по правде говоря, кому это интересно? Открою маленький секрет: наша стратегия для CityDesk – не допустить массивного пиара до выпуска версии 2.0. Это и будет *та* версия, которая произведет на всех впечатление. А пока что мы ведем тихий партизанский маркетинг, и тот, кто на нее наткнется, обнаружит, что это классная программа, которая решает многие его проблемы, но Арнольду Тойнби (Arnold Toynbee) не придется ничего переписывать.

Для коммерческого ПО процесс разработки, прототипирования, интеграции, исправления ошибок, полного цикла альфа- и бета-тестирования, создания документации и так далее обычно длится 6–9 месяцев. Фактически, если вы захотите каждый год делать полный новый выпуск, у вас остается меньше 3 месяцев на разработку нового кода. Ежегодно обновляемое программное обеспечение обычно не воспринимается как обладающее достаточным для новой версии числом новых функциональных возможностей. (Corel PhotoPaint и Intuit Quickbooks – наглядные тому примеры; каждый год выпускаются их новые «большие» версии, приобретение кото-

рых редко бывает оправданным.) В результате многие пользователи привыкают пропускать каждый второй релиз. Едва ли вы хотите, чтобы у ваших пользователей сложилась подобная привычка. Если вы растянете план до года-полутора между выпусками, то получите шесть месяцев на создание новых функциональных возможностей вместо трех, в результате клиент может оказаться более склонным к обновлению своей версии.

Ладно, если пятнадцать месяцев – это хорошо, может быть, два года – еще лучше? Может быть. Некоторым компаниям это сходит с рук, если они лидеры в своей категории. Например, разработчикам Photoshop. Но как только приложение начнет выглядеть устаревшим, его перестанут покупать, ожидая выпуска новой версии со дня на день. Для софтверного бизнеса это может стать серьезной потерей дохода. И, конечно же, у вас могут быть конкуренты, наступающие вам на пятки.

Для больших программных платформ – операционных систем, компиляторов, веб-браузеров, СУБД – наибольшую трудность в разработке составляет обеспечение совместимости с тысячами и даже миллионами существующих программных или аппаратных продуктов. При выходе новой версии Windows проблемы обратной совместимости крайне редки. Достигается это ценой безумного объема тестирования, в сравнении с которым Панамский канал – просто самоделка на один уикенд. В обычном трехлетнем цикле выпуска основных версий Windows большая часть времени тратится на утомительные этапы интеграции и тестирования, а не на создание новых функций. Выпускать новые версии чаще нереально, люди просто сойдут с ума. Сторонние разработчики программного и аппаратного обеспечения забастуют, если им придется тестировать множество маленьких релизов операционной системы. **Для систем с миллионами пользователей и миллионами точек интеграции предпочтительнее редкие выпуски.** Поступайте, как Apache: один релиз в начале интернет-пузыря, и один – в конце. Идеально.

Если проводить много проверок и поблочных тестов и писать свои программы аккуратно, можно приблизиться к тому, чтобы каждая ежедневная сборка обладала *почти* достаточным качеством для выпуска. К этому, несомненно, нужно стремиться. Даже если вы планируете следующий выпуск только через три года, обстановка на рынке может неожиданно измениться, и окажется разумным выпустить промежуточную версию, чтобы ответить на действия конкурента. Когда жена на сносях, не стоит разбирать автомобиль. Лучше параллельно вести новую версию, не переходя на нее, пока она не достигнет совершенства.

Но не переоценивайте значение высококачественных ежедневных сборок. Количество ошибок может постоянно держаться на нулевом уровне, но если ваша программа должна выйти в самостоятельную жизнь, невозможно выявить потенциальные ошибки совместимости, ошибки Windows 95 и ошибки типа «не работает при включении крупных шрифтов», не выпустив несколько бета-версий, а провести бета-тестирование быстрее, чем за восемь недель, нереально.

И последнее. Если ваша программа поставляется как веб-сервис, вроде eBay или PayPal, то, теоретически, ничто не мешает вам часто выпускать версии с небольшими изменениями, хотя, возможно, это и не лучший вариант. Помните главное правило юзабилити: приложение удобно, если оно ведет себя так, как того *ожидает* пользователь. Если каждую неделю что-то менять, приложение будет не столь предсказуемым, и, следовательно, не таким удобным. (И не рассчитывайте на надоедливую надпись «Внимание! Интерфейс изменился!» – ее никто не читает.) С позиции юзабилити лучше выпускать новые версии реже, вводя сразу целый пакет модификаций, изменяющих внешний вид сайта на непривычный, – так пользователи интуитивно поймут, что все существенно изменилось и нужно проявлять осторожность.

ГЛАВА ТРИДЦАТЬ ЧЕТВЕРТАЯ

Верблюды и резиновые уточки

15 ДЕКАБРЯ 2004 ГОДА, СРЕДА

Вы только что выпустили свежую версию программы для работы с фотоальбомами. В качестве упражнения читателю предлагается разработать механизм оповещения пользователей об этом событии. Может, у вас есть популярный блог или еще что-то. А может, Уолт Моссберг написал восторженную рецензию на вашу программу в «Wall Street Journal».

Один из главных вопросов здесь – назначение цены для новой программы. Эксперты не знают, как на него ответить. «Ценообразование – тайна, покрытая мраком», – скажут они вам. Самая большая ошибка софтверных компаний – занижение цены, результат которого – недостаток прибыли и уход из бизнеса. Но еще большая ошибка – да-да, больше, чем самая большая, – завышение цены, результат которого – недостаток покупателей и уход из бизнеса. Уходить из бизнеса плохо, потому что все теряют работу, и вы нанимаетесь в супермаркет зазывалой с минимальным окладом и униформой из синтетики.

Так что, если вам больше нравится униформа из хлопка, стоит разобраться в данном вопросе.

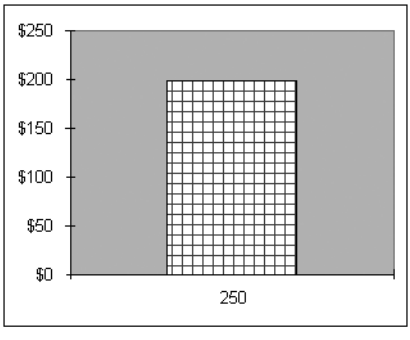
Ответ действительно непрост. Сначала я вкратце изложу экономическую теорию, а затем расправлюсь с ней – в итоге вы будете гораздо более сведущи в ценообразовании, по-прежнему не зная, сколько просить за свою программу, но такова природа ценообразования. Если вам лень читать дальше, продавайте программу по 0,05 доллара (а если она предназначена для отслеживания ошибок, то заломите 30 000 000 долларов).

Так, о чем это я?

Немного экономической теории

Допустим, ваша программа стоит 199 долларов. Почему 199? Просто нужно с чего-то начать. Скоро появятся и другие числа. Но пока будем исходить из того, что программа стоит 199 долларов, и по этой цене ее готовы приобрести 250 покупателей.

Вот иллюстрация:



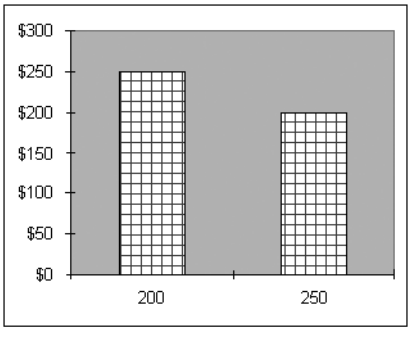
Этот небольшой график показывает, что если ПО стоит 199 долларов, то его купят 250 человек. (Как видите, экономисты – довольно странные люди – они предпочитают откладывать количество продаж по оси *X*, а цену продукта – по оси *Y*. Если 250 человек купили вашу программу, значит, вы оценили ее в 199 долларов!)

Что произойдет, если вы поднимете цену до 249 долларов?

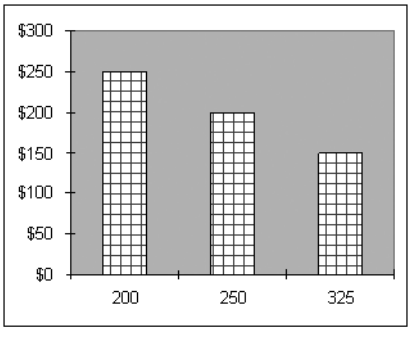
Некоторые из тех, что были готовы заплатить 199 долларов, решат, что 249 долларов – слишком дорого, и откажутся от покупки.

Очевидно, что те, кто не собирался платить за продукт 199 долларов, не купят его и по более высокой цене.

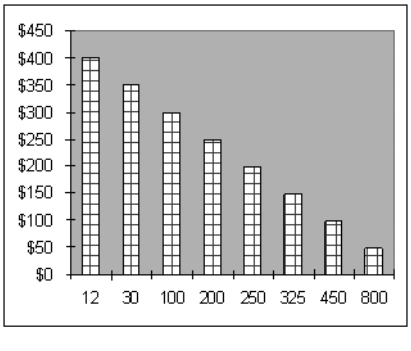
Если 250 человек купили продукт за 199 долларов, мы должны предположить, что за 249 долларов его приобретут *меньше* покупателей. Скажем, человек 200:



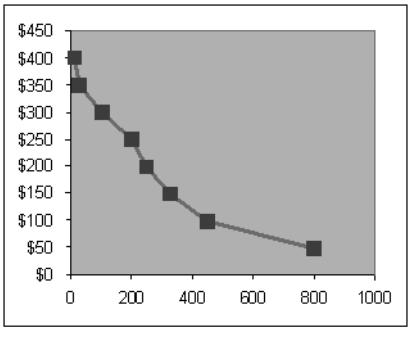
А что если назначить *меньшую цену*, например 149 долларов? Тогда все, кто был готов купить продукт за 199 долларов, наверняка купят его за 149 долларов. И появятся еще те, кто решится потратить 149 долларов, так что при цене 149 долларов мы продадим, скажем, 325 экземпляров:



И так далее:



Вместо отдельных точек нарисуем кривую, которая через них проходит, а заодно зададим масштаб по оси *X*.



Теперь называйте любую цену от 50 до 400 долларов, а я скажу, сколько человек купит вашу программу по этой цене. Мы сейчас нарисовали классическую *кривую спроса*. Кривая спроса всегда убывающая, поскольку чем дороже продукт, тем меньше желающих его приобрести.

Эти цифры, конечно, взяты с потолка. Прошу поверить мне на слово лишь в одном: кривая спроса всегда убывающая.

(Если вас все еще раздражает, что я откладываю количество продаж по оси *X*, а цену по оси *Y*, когда очевидно, что количество является функцией цены, а не наоборот, то все претензии к Огюстену Курно (Augustin Cournot). Возможно, он уже завел блог.)

Так какую же цену назначить?

«Нуу... скажем, 50 долларов – ведь тогда я продам больше экземпляров!»

Нет-нет, ваша задача – максимизировать не *количество продаж*, а *прибыль*.

Давайте посчитаем прибыль.

Предположим, каждый продаваемый экземпляр программы стоит для вас **дополнительные** 35 долларов.

Пусть на разработку ПО вы потратили 250 000 долларов – это так называемые невозвратные капиталовложения. Нас больше не интересует эта сумма, потому что она не изменится, продадите вы 1000 экземпляров или ни одного. Невозвратные. Попрощайтесь с ними. Какую бы цену вы ни назначили, 250 000 долларов ушли, и больше о них не вспоминайте.

В данный момент вас должны интересовать дополнительные издержки продажи *каждого нового экземпляра*. Они могут включать в себя доставку

и обработку, техподдержку, стоимость банковских услуг, печати дисков, упаковки и так далее. 35 долларов я взял с потолка.

Запускаем нашу замечательную электронную таблицу:

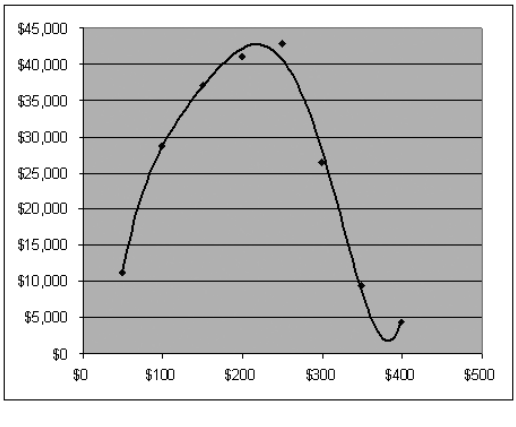
	A	B	C	D	E
1	Quantity	Price	Incr. Cost	Unit Profit	Total Profit
2				(Price - Incr Cost)	(Unit Profit x Quantity)
3	12	$399	$35	$364	$4,368
4	30	$349	$35	$314	$9,420
5	100	$299	$35	$264	$26,400
6	200	$249	$35	$214	$42,800
7	250	$199	$35	$164	$41,000
8	325	$149	$35	$114	$37,050
9	450	$99	$35	$64	$28,800
10	800	$49	$35	$14	$11,200
11					

Читать эту таблицу нужно так. Каждая строка – это сценарий. Строка №3: *Если* цена (Price) 399 долларов, *то* продаем 12 экземпляров (Quantity), прибыль *с каждого* экземпляра (Unit Profit) 364 долларов, что в сумме дает 4 368 долларов общей прибыли (Total Profit) при дополнительных издержках (Incr. Cost) 35 долларов на экземпляр.

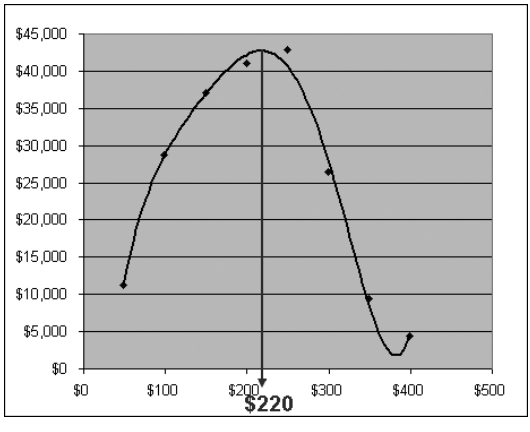
ЭТО УЖЕ ЧТО-ТО!

И впрямь здорово, ведь мы вот-вот решим задачу выбора оптимальной цены для нашего продукта. Я ТАК ВОЛНУЮСЬ!

И вот почему. Если мы нарисуем график прибыли в зависимости от цены, то получим красивую кривую с большим горбом посередине. И всем понятно, что значит этот горб. Горбы – это локальные максимумы! Или верблюды. В данном случае это именно локальный максимум.



В этом графике реальные данные отображены ромбиками, и я попросил Excel нарисовать по ним красивую полиномиальную линию тренда. Остается опустить перпендикуляр из вершины горба, и я узнаю цену, дающую максимальную прибыль:



«О храброславленный герой! Хвалу тебе пою!» – ликую я. Мы нашли оптимальную цену – 220 долларов, по этой цене вам и надо продавать свою программу.

Спасибо за внимание! Представление окончено! До новых встреч!

Вы все еще здесь?

Понимаю.

Некоторые, самые наблюдательные читатели догадались, что у меня есть что еще сказать, кроме «220 долларов».

Что ж, пожалуй. Кое-что упущено и не завершено, и я продолжу, если не возражаете. Нет? Хорошо.

Дело в том, что, назначив цену 200 долларов, мы смогли бы продать, скажем, 233 экземпляра, получив общую прибыль 43 105 долларов. Замечательно, но меня кое-что смущает: с тех, кто готов заплатить больше, как эти 12 человек, готовые отдать 399 долларов, мы взяли бы по 220 долларов, как и со всех остальных!

Разница между 399 и 220 долларами (то есть 179 долларов) называется *выгодой потребителя* (consumer surplus). Это сумма, которую богатые покупатели сэкономили на своей покупке, без чего они вполне могли обойтись.

Это как если бы, собираясь купить в Banana Republic шерстяной свитер за 70 долларов (вполне достойная цена), вы попали на распродажу и приобрели его всего за 50. Вы нашли на дороге 20 долларов, которые были готовы отдать банановцам!

Во как!

Хорошего капиталиста это раздражает. Елки-палки, если ты *хочешь* расстаться с 20 долларами – отдай их мне! Я найду им применение – куплю джип, дом, самолет, яхту – что там еще покупают капиталисты?

На экономическом сленге это называется *захватить выгоду потребителя*.

Давайте поступим так. Вместо того чтобы брать 220 долларов, спросим каждого покупателя, богат он или беден. Если он скажет, что богат, возьмем с него 349 долларов, а если беден, то 220.

Сколько теперь получится? Снова в Excel:

	A	B	C	D	E
1	Customers	Price	Qty Sold	Unit Profit	Total Profit
2	Rich	$349	42	$314	$13,188
3	Poor	$220	191	$185	$35,335
4				**TOTAL:**	**$48,523**

Обратите внимание на количественные показатели: мы продадим все те же 233 экземпляра (Qty Sold), но 42 самым богатым покупателям (Customers: Rich), готовым расстаться с 349 долларами, предложим потратить 349 долларов (Price). При разной прибыли с экземпляра (Unit Profit) наша общая прибыль (Total Profit) возрастет с 43 K до 48 K долларов. ЗДОРОВО!

Дайте мне еще этой выгоды потребителя!

Постойте, это еще не все. 233 копии продано, но мне неудобно перед теми, кто не может заплатить больше 99 долларов. В конце концов, продав еще несколько экземпляров этим ребятам по 99 долларов, я все равно *заработаю*, раз маргинальная стоимость у меня всего 35 долларов.

Что если позвонить всем клиентам, отказавшимся покупать программу за 220 долларов, и предложить им ее за 99 долларов?

На цену в 99 долларов есть 450 потенциальных покупателей – не забывайте, что 233 из них заплатят полную цену, так что остаются 217 дополнительных покупателей, которым полная цена не по карману:

	A	B	C	D	E
1	Customers	Price	Qty Sold	Unit Profit	Total Profit
2	Rich	$349	42	$314	$13,188
3	Poor	$220	191	$185	$35,335
4	Callback & Beg	$99	217	$64	$13,888
5				**TOTAL:**	**$62,411**

Мама дорогая, похоже, мы срубили 62 К долларов прибыли! Порядка двадцати тысяч свободных денег – вот и первый взнос за рыболовный катер, на который давно положен глаз! Вот что дает **сегментирование** – деление покупателей по категориям платежеспособности и получение максимальной выгоды потребителя от каждого покупателя. Святые Сегменты, Бэтмен, сколько бы мы заработали, если б могли заставить каждого покупателя *назвать* максимальную сумму, с которой он готов расстаться, и взять с него именно эту цену!

	A	B	C	D
1	Qty Sold	Price	Unit Profit	Total Profit
2	12	$ 399	364	$ 4,368
3	18	$ 349	314	$ 5,652
4	70	$ 299	264	$ 18,480
5	100	$ 249	214	$ 21,400
6	50	$ 199	164	$ 8,200
7	75	$ 149	114	$ 8,550
8	125	$ 99	64	$ 8,000
9	350	$ 49	14	$ 4,900
10			**TOTAL**	**$ 79,550**

Невероятно! Почти 80 К. Почти вдвое больше, чем при единой цене. Загребать выгоду потребителя однозначно прибыльно. Даже с тех досадных 350 покупателей, не желающих платить за мою программу больше 49 долларов, можно что-то взять. И все эти потребители счастливы, потому что мы предлагаем им заплатить ту цену, на которую они сами согласны, так что никто никого не грабит. В известном смысле.

Вот несколько, возможно, знакомых вам примеров сегментирования:

- Скидки для пожилых: большинство из них живут на «фиксированный доход» и склонны платить меньше, чем люди в работоспособном возрасте.
- Дешевые дневные киносеансы (на них ходят те, кто не работает).
- Странные тарифы авиакомпаний, где все платят по-разному. Секрет в том, что командированным перелет оплачивают их компании, поэтому им совершенно безразлично, сколько стоит билет, тогда как

путешествующие в свое удовольствие тратят собственные деньги, и они просто не полетят, если билеты будут слишком дорогими.

Разумеется, авиалинии не могут просто *спросить* у вас, по делу вы летите или нет, потому что все быстро разгадают этот секрет и станут обманывать, чтобы летать дешевле. Но командированные, как правило, летают в будни и не любят проводить выходные вне дома. Поэтому авиакомпании исповедуют принцип «остается на субботу – значит, не командированный», продавая билеты дешевле тем, кто задерживается на субботнюю ночь.

Есть и более тонкие способы сегментирования. Например, газетный купон на скидку – тот, за который получаешь 25 центов скидки с пачки стирального порошка, если вырежешь его из газеты и принесешь в магазин. Фишка этих купонов в том, что их вырезание, сортировка, хранение, запоминание и поиск брендов, на которые у вас есть купоны, – ваш ручной труд за 7 долларов в час или около того.

Для живущего на пособие пенсионера 7 долларов в час – хорошая приманка, и он будет вырезать купоны. А для биржевого аналитика из Merrill Lynch, который получает 12 000 000 долларов в год за хорошие отзывы о паршивых интернет-компаниях, 7 долларов в час – просто смех, и он не станет вырезать купоны. Ха, да за час он может раз *десять* ответить «покупайте» на вопрос о какой-нибудь паршивой интернет-компании! Так что купоны – один из способов для компаний назначить две разные цены на товар, фактически сегментируя рынок надвое. Примерно то же самое – купон на скидку, получаемый по почте в ответ на запрос, только при этом вы сообщаете свой почтовый адрес и в дальнейшем можете стать объектом прямого маркетинга.

Способов сегментирования рынка много. Можно предлагать свой продукт под разными торговыми марками («Old Navy», «Gap», «Banana Republic») в расчете на то, что богатые привыкли покупать одежду в Banana Republic, в то время как бедные ходят в Old Navy. И чтоб никто не перепутал, магазин Banana Republic открывают рядом с кондоминиумом по 2 000 000 долларов, а Old Navy – возле вокзала, куда вы притаскиваете свою усталую задницу, чтобы вернуться в Нью-Джерси после тяжелого трудового дня.

В мире ПО можно одну версию своего продукта назвать Professional, а другую, почти такую же, – Home, чтобы корпоративные покупатели (опять-таки, те, кто тратит чужие деньги), стыдясь использовать «Windows XP Home Edition» *на работе*, купили версию Pro. Домашняя версия в офисе? Это как прийти на работу в пижаме.

Простой прием: если хотите заняться сегментированием, попробуйте ввести не повышенный тариф, а *скидку* для определенных пользователей. Никто не хочет чувствовать себя обобранным: люди скорее купят за 99 долларов то, что стоит 199 долларов, чем за 79 то, что стоит 59. Казалось бы, человек должен подойти рационально: 79 долларов меньше, чем 99. Но на самом деле ему не нравится мысль, что на нем кто-то наживается. Приятнее думать о выгодной покупке, а не о том, что тебя поимели.

ТАК ИЛИ ИНАЧЕ.

До этого момента все было просто.

Сложность в том, что все вышесказанное не совсем верно.

Вернемся к затее с сегментированием. Она раздражает людей. Люди хотят платить цену, которая кажется им справедливой. Им не нравится думать, что с них берут лишнее только потому, что они недостаточно умны, чтобы найти волшебный купон. Авиакомпании настолько преуспели в сегментировании, что буквально для каждого пассажира – своя цена. В результате многие, осознав, что платят дороже, возненавидели эти авиалинии. И когда появилась альтернатива в виде дешевых авиаперевозчиков, у пассажиров не осталось никакой лояльности к старым авиалиниям, все эти годы пытавшимся их обирать.

И бог вам в помощь, если какой-нибудь популярный блогер обнаружит, что дорогая модель вашего принтера отличается от дешевой только отсутствием ограничителя скорости.

Так что сегментирование, хотя и помогает «захватить выгоду потребителя», может иметь и негативные последствия для имиджа вашего продукта в долгосрочном плане. Многие небольшие разработчики ПО обнаружили, что их прибыль выросла, а количество торгующихся покупателей значительно уменьшилось, когда они отказались от купонов, скидок, нескольких версий и уровней. Похоже, покупатель скорее готов заплатить 100 долларов наравне с другими, чем 79 долларов, если кто-то заплатил 78. Да что там, General Motors создала *целую автомобильную компанию* Saturn, построив ее на том принципе, что предложенная цена справедлива, и вы не будете торговаться.

Даже не боясь утратить в долгосрочной перспективе свой гудвил из-за явной ценовой дискриминации, осуществить сегментирование все равно не так-то просто. Прежде всего, как только ваши покупатели поймут, что вы этим занимаетесь, они начнут выдавать себя за других:

- Те, кто часто летает по делам, станут брать билеты с двойной субботней ночевкой. Например, консультант из Питтсбурга, работающий в Сиэтле с понедельника по четверг, покупает *двухнедельную поездку* из Питтсбурга в Сиэтл, в середине которого покупает билеты для полета домой на уикенд. Оба путешествия включают субботнюю ночевку – рейсы те же самые, но цена гораздо ниже.
- У вас есть академическая скидка? Каждый, имеющий хотя бы *отдаленное* отношение к кому-то, *отдаленно* связанному с образованием, будет требовать эту скидку.
- Если пользователи общаются между собой, они могут выяснить, какие цены вы предлагаете другим, и вам придется снизить цену до минимальной для всех. *Особенно* это относится к тем, кто делает закупки для крупных корпораций и теоретически должен проявлять максимальную «готовность платить», поскольку представляет богатого клиента. У корпораций есть специальные департаменты закупок, где люди *получают зарплату* за то, что добиваются самых выгодных цен. Эти люди посещают *конференции,* где их учат, как это делается. Они *целый день перед зеркалом* репетируют фразу «Нет. Дешевле». У вашего парня, который занимается продажами, просто нет шансов.

Пронырливые компании-разработчики софта применяют две формы сегментирования, на практике оказывающиеся не слишком хорошими идеями:

Плохая идея № 1: неограниченные лицензии

На самом деле, это противоположность сегментированию. Некоторые мои конкуренты практикуют следующее: берут с небольших клиентов плату по количеству пользователей системы, одновременно предлагая «неограниченную» лицензию по фиксированной цене. Это глупо, потому что самую значительную скидку вы даете *именно* крупнейшим покупателям – тем, кто готов заплатить вам *больше всех.* Вы действительно хотите, чтобы IBM приобрела ваше ПО для своих 400 000 сотрудников за 2000 долларов?

Вводя «неограниченную лицензию», вы сразу дарите гигантский объем выгоды потребителя наименее чувствительным к цене клиентам, которые должны были бы стать дойной коровой для вашего бизнеса.

Плохая идея № 2: ценообразование по принципу «А сколько у вас есть?»

Это способ, применяемый в стартапах, создаваемых бывшими менеджерами Oracle, когда нигде на веб-сайте не указана цена. Можете искать сколько угодно – найдете только форму, где вас просят оставить ваше имя, адрес, номер телефона и факса, хотя отправлять вам факсом они ничего не собираются.

Идея, очевидно, в том, чтобы с вами связался менеджер по продажам, чтобы выяснить, сколько вы стоите, и выставить вам счет на соответствующую адекватную сумму. Идеальное сегментирование!

Здесь тоже не все в порядке. Во-первых, бюджетные покупатели пройдут мимо. Они решат, что раз цена не указана, значит она высока и они ее «не потянут». Во-вторых, противники общения с напористыми менеджерами по продажам также пройдут мимо.

Хуже то, что как только вы дадите понять, что о цене можно договориться, произойдет *обратное сегментирование.* И вот почему. Ваши покупатели – крупные компании, предположительно легко расстающиеся с деньгами, – весьма искушены в вопросах закупок. Они поймут, что ваш менеджер по продажам работает на комиссионных, и они знают, что у него есть квартальный план. Они также знают, что и он сам, и его компания отчаянно нуждаются в продажах в конце квартала (менеджер – чтобы получить свои комиссионные, компания – чтобы избежать расправы со стороны инвесторов). Поэтому большие корпорации всегда будут ждать до последнего дня квартала, чтобы получить неприлично низкую цену, при этом вам придется с помощью неких бухгалтерских ухищрений провести по книгам компании огромные поступления, которых в реальности никогда не будет.

Так что не надо предлагать неограниченные лицензии и не надо придумывать цены на ходу.

НО ПОСТОЙТЕ!

Вы *действительно* хотите максимизировать прибыль? Я что-то упустил. Вас ведь, наверно, интересует прибыль не только в текущем месяце. На самом деле, вы хотите максимизировать все прибыли, включая будущие. Выражаясь формально, вы хотите увеличить NPV (чистую приведенную стоимость) потока всех будущих прибылей (так, чтобы резервы наличности никогда не опускались ниже нуля).

Для справки: что такое NPV?

Что стоит больше, 100 долларов сегодня или 100 долларов через год?

Очевидно, 100 долларов сегодня, потому что вы можете инвестировать их, скажем, в облигации, и в конце года у вас будет 102,25 доллара.

Так что если вы собираетесь сравнивать стоимость 100 долларов через год и сегодня, вам нужно *дисконтировать* эту сумму, исходя из некоторой процентной ставки. Если процентная ставка составляет, скажем, 2,25%, то эти будущие 100 долларов следует уценить до 97,80 доллара, что называется *чистой приведенной стоимостью* (net present value, NPV) 100 долларов через один год.

Чем дальше отстоит срок, тем сильнее следует дисконтировать. 100 долларов через пять лет, по текущим процентным ставкам, сегодня стоят только 84 доллара. 84 доллара – это чистая приведенная стоимость 100 долларов через пять лет.

Сколько бы вы предпочли заработать?

Первый вариант: 5000 долларов, 6000 долларов, 7000 долларов в течение следующих трех лет.

Второй вариант: 4000 долларов, 6000 долларов, 10 000 долларов в течение следующих трех лет.

Вариант номер два кажется более выгодным, даже если дисконтировать будущие прибыли. Во втором варианте получается так, как если бы вы инвестировали 1000 долларов в первый год и получили 3000 долларов через два года – неплохое капиталовложение!

Я поднял этот вопрос, потому что по ценам программное обеспечение делится на три вида: бесплатное, дешевое и дорогое.

1. **Бесплатное.** Open source и так далее. Не относится к нашей дискуссии. Ничего интересного. Идем дальше.
2. **Дешевое.** От 10 до 1000 долларов, продается множеству покупателей по низкой цене, без специалистов по продаже. Большинство коробочных продуктов для отдельных потребителей и мелкого бизнеса попадает в этот раздел.

3. **Дорогое**. От 75 000 до 1 000 000 долларов, продается нескольким богатым компаниям с помощью команды ловких торговых агентов, которые ради одной продажи шесть месяцев интенсивно проводят презентации PowerPoint. Модель Oracle.

Все три метода работают.

Заметили разрыв в ценах? Нет программ стоимостью от 1000 до 75 000 долларов. Объясню, почему. Как только вы переходите барьер 1000 долларов, вам нужна *серьезная* работа с корпорациями. Вам нужна строка в их бюджете. Вам нужно добиться одобрения менеджерами по закупкам и высшим руководством, победить конкурирующие предложения и выполнить массу бумажной работы. Поэтому вам придется направить в эту корпорацию профессионального менеджера по продажам, вооруженного PowerPoint, оплатить его бизнес-класс в самолете, членство в гольф-клубе и порнофильмы по 19,95 долларов в отеле Ritz Carlton. С учетом этого стоимость одной успешной продажи должна быть в среднем около 50 000 долларов. Отправляя к своим клиентам торговых агентов и беря за свой продукт меньше 75 000 долларов, вы окажетесь в убытке.

Самое смешное то, что крупные компании настолько серьезно оберегают себя от дорогих покупок, что практически взвинчивают стоимость дорогого ПО с 1000 до 75 000 долларов. Большая часть этих денег идет на преодоление тех барьеров, которые они возвели, чтобы обезопасить себя от ошибочных покупок.

Достаточно взглянуть на сайт Fog Creek, чтобы убедиться: я прочно закрепился в лагере номер два. Почему? Продажа ПО по низкой цене позволяет сразу получить тысячи клиентов, как мелких, так и крупных. И все они будут использовать мой продукт и рекомендовать его своим друзьям. По мере своего роста эти клиенты станут покупать дополнительные лицензии. Когда люди, работающие в этих компаниях, перейдут в другие, они станут рекомендовать мой продукт на новом месте. Фактически я соглашаюсь на текущую низкую цену в обмен на получение поддержки рядовых работников. Я рассматриваю низкую цену FogBugz как расходы на рекламу, которые, по моим расчетам, должны многократно окупиться в долгосрочной перспективе. Пока что эта схема работает замечательно: продажи FogBugz выросли более чем на 100% за три года без маркетинговых усилий, исключительно на устных рекомендациях и на покупке дополнительных лицензий имеющимися клиентами.

Возьмем для сравнения BEA. Большая компания. Высокий уровень цен. Сама цена уже означает, что практически никто не работал с их продуктом.

Никто по окончании колледжа не начинает новый бизнес на основе технологии BEA, потому что продукты BEA не были доступны в колледже. Множество других замечательных технологий похоронили себя высокими ценами: Apple WebObjects не рассматривался как сервер приложений, потому что его начальная цена была 50 000 долларов. Кому было интересно, насколько он хорош? Никто его даже в глаза не видел. Как и все, что создано в Rational. Попасть к пользователю такие продукты могут только в результате дорогостоящей массированной коммерческой кампании. При таких ценах кампания направлена на обработку руководства, а не технических специалистов. Технические специалисты могут активно сопротивляться насильно навязываемым им руководством плохим технологиям с хорошо организованными продажами. Среди покупателей FogBugz предостаточно тех, у кого на полках пылятся продукты Remedy, Rational или Mercury, за которые было выложено намного больше 100 000 долларов, потому что это ПО не годится для практической работы. Потом они покупают FogBugz за пару тысяч долларов и начинают реально пользоваться этим продуктом. Менеджер по продажам компании Rational смеется мне в лицо, потому что у меня 2000 долларов в банке, а у него 100 000. Но у меня куда больше клиентов, чем у него, и все они пользуются моим ПО, и агитируют за него, и распространяют его, в то время как клиенты Rational либо а) не пользуются им, либо б) пользуются, но терпеть его не могут. Но он все равно смеется надо мной на своей 40-футовой яхте, в то время как я играю с резиновыми уточками в ванне. Как я сказал, *все три метода действуют хорошо*. Но низкая цена – все равно что покупка рекламы, а значит, инвестиции в будущее.

Так.

О чем это я говорил?

Ах, да, прежде чем начать доказывать с пеной у рта, я разбирал логику получения кривой спроса. Во время всего этого обсуждения кривой спроса вы наверняка спрашивали себя: «Как же узнать, сколько люди готовы заплатить за мой софт?»

Вы правы.

Это проблема.

На самом деле, построить кривую спроса нельзя.

Можно организовать фокус-группы и опрашивать людей, но они не скажут вам правду. Одни будут врать, чтобы показать свою щедрость и состоятельность. «Без базара, я покупаю пару джинсов за 400 долларов в New York Minute». Другие будут врать, потому что действительно нуждаются

в вашем товаре, и им кажется, что если они назовут меньшую сумму, вы снизите цену. «Софт для блога? Ну, 38 центов я за него еще заплачу».

На следующий день вы опрашиваете другую фокус-группу, первый из выступающих в которой, запав на симпатичную соседку, захотел поразить ее рассказом о своей дорогой машине, после чего все стали оперировать Большими Числами. А еще через день во время перерыва вы предлагаете всем пойти в буфет, и там они обсуждают, не слишком ли дорого 4 доллара за чашку кофе, но вы этого не слышите, потому что вышли в туалет, а позже, когда вы начинаете выяснять, сколько же они готовы заплатить, все вдруг становятся очень бережливыми.

Затем фокус-группа все-таки соглашается, что ваш продукт стоит 25 долларов в месяц, но когда вы спрашиваете, сколько бы они заплатили за бессрочную лицензию, те же люди не дают ни цента сверх 100 долларов. Они просто *не умеют* считать.

Или вы спрашиваете у авиаконструкторов, сколько бы они заплатили, и те отвечают, что максимум 99 долларов, хотя каждый день пользуются программой за 3000 долларов в месяц, даже не зная об этом, потому что закупками занимаются совсем другие люди.

В результате изо дня в день на вопрос «сколько вы готовы заплатить за то-то?» вы будете получать абсолютно – подчеркиваю, *абсолютно* – разные ответы. Правда в том, что единственный способ определить, сколько люди готовы платить за товар, – выставить его на продажу и просто посмотреть, сколько человек его купит.

Можно также поиграть с ценами, чтобы измерить чувствительность рынка к цене и попытаться выявить кривую спроса. Но если у вас нет 1 000 000 клиентов и абсолютной уверенности, что клиент *A* никогда не узнает, что вы предложили клиенту *B* более выгодную цену, то вы не получите статистически достоверные результаты.

Есть устойчивое мнение, что опыты на многих людях – примерно то же, что школьные опыты на уроках химии. Но каждый, кто экспериментировал на людях, знает, что результаты оказываются очень разными, воспроизвести их невозможно, и единственный способ сохранить уверенность в результате – не проводить один и тот же эксперимент дважды.

На самом деле, нельзя даже уверенно утверждать, что кривая спроса – убывающая.

Единственное основание для вывода о том, что эта кривая убывающая, – предположения типа «если Фредди хочет купить пару кроссовок

за 130 долларов, то он, несомненно, с радостью купит их за 20 долларов». Верно? Как бы не так! Уж во всяком случае, если Фредди – американский тинейджер. Американский тинейджер умрет, но не выйдет на люди в кроссовках за 20 долларов. Это же смертный приговор! Кроссовки за двадцать баксов?! В школе?!!

И я не шучу: цена посылает сигнал. Билет в кино в моем городе, стоит, кажется, 11 долларов. Однако в одном кинотеатре брали 3 доллара за вход. Ходил ли туда кто-нибудь? **НЕ ДУМАЮ.** Ведь было очевидно, что это помойка с плохими фильмами. А тот, кто осмелился сказать публике, какие фильмы действительно отстойные, отправился на дно реки в бетонных кроссовках за 20 долларов.

Дело в склонности людей считать, что сколько они заплатили, столько и получат взамен. Когда какое-то время назад мне понадобилось много дисковой памяти, я вложился в замечательные дешевые диски, якобы спроектированные лично господином Порше, заплатив около 1 доллара за гигабайт. За полгода все четыре диска умерли. На прошлой неделе я заменил их на SCSI-диски Seagate Cheetah, примерно 4 доллара за гигабайт, потому что пользуюсь такими дисками с момента основания своей компании уже четыре года без всяких проблем. Это в подтверждение тезиса «вы получаете то, за что платите».

Есть масса примеров тому, что вы *действительно* получаете то, за что платите, и средний покупатель считает лучшим тот продукт, который дороже. Покупаете кофеварку? Хотите *действительно* хорошую кофеварку? Есть два варианта – пойти в библиотеку и почитать отзывы покупателей в журнале либо пойти в Williams-Sonoma и купить там самую дорогую кофеварку.

Устанавливая цену, вы посылаете сигнал. Если ПО вашего конкурента стоит от 100 до 500 долларов и вы решаете оценить свой продукт в 300 долларов, поскольку он «еще не дозрел», то какое сообщение, по-вашему, вы посылаете покупателю? Вы говорите ему: «Мой софт – так себе». У меня есть идея получше: просите за него 1350 долларов. Тогда ваши клиенты будут думать, что это *нехилая штука*, раз за нее хотят такие *бешеные бабки*.

Но они не купят ее, потому что лимит корпоративной карты AMEX – 500 долларов.

Жаль.

Чем больше вы изучаете ценообразование, тем меньше о нем знаете.

Я натрепал на эту тему уже больше 5000 слов, и все еще не чувствую, что мы к чему-то пришли – вы и я.

Иногда я думаю о том, как хорошо работать в такси, потому что цены там закреплены законом. Или продавать сахар. Просто сахар. Да. Это была бы сладкая жизнь.

В общем, как я и советовал, просите за свою программу 0,05 доллара. Но если она отслеживает ошибки, то правильная цена – 30 000 000 долларов. Спасибо за внимание, и простите, что оставляю вас в еще большем неведении относительно правильного выбора цены для своего продукта, чем до чтения этой главы.

ЧАСТЬ ДЕВЯТАЯ

Ревизия программного продукта

ГЛАВА ТРИДЦАТЬ ПЯТАЯ

Пять почему

22 ЯНВАРЯ 2008 ГОДА, ВТОРНИК

10 января 2008 года в пол-четвертого ночи наш системный администратор Майкл Горсач (Michael Gorsuch) проснулся у себя дома в Бруклине от резкого звука. Это программа Nagios, контролирующая нашу сеть, прислала текстовое сообщение о возникшей неисправности.

Он встал с постели, наступив на собаку (чем ее разбудил), та в сердцах сделала лужу в коридоре и вернулась на лежанку. Майкл же, сев за компьютер в соседней комнате, обнаружил, что один из трех наших центров обработки данных, в центре Манхэттена, недоступен из Интернета.

Этот центр расположен в охраняемом здании в центре Манхэттена, специальным оборудованием управляет интернет-провайдер PEER 1. Там есть резервные генераторы с запасом дизельного топлива на несколько дней и многочисленные аккумуляторы для питания всего оборудования в течение нескольких минут, пока запускаются генераторы. Там есть мощные воздушные кондиционеры, многократно дублирующиеся высокоскоростные соединения с Интернетом и «правильные» инженеры-практики из тех, кто все делает с чувством, с толком, с расстановкой, без модного экстрима, – словом, все очень надежно.

Такие интернет-провайдеры, как PEER 1, обычно гарантируют безотказную работу на условиях SLA (Service Level Agreement – соглашение об уровне обслуживания). В типичном SLA может быть указано, к примеру, что время безотказной работы составляет 99,99%. Арифметика говорит, что в году 525 949 минут (или 525 600, как считают в мюзикле «Rent»), зна-

чит допустимый простой за год – 52,59 минуты. При большем простое SLA обычно предусматривает какие-то штрафные санкции, по правде говоря, смешные – например, возврат денег за те минуты, когда оборудование было неисправно. Однажды за двухдневное отключение, стоившее нам тысяч долларов, мы получили от провайдера T1 скидку порядка 10 долларов. В этом отношении SLA довольно бессмысленны, и с учетом незначительности штрафных санкций многие провайдеры сетевых услуг стали рекламировать 100% времени безотказной работы.

Через 10 минут все заработало, и Майкл снова лег.

Примерно до 5:00. На этот раз Майкл позвонил в центр управления сетями PEER 1 (Network Operations Center, NOC) в Ванкувере. Они что-то протестировали, провели расследование, не выявив никаких неисправностей, и к 5:30 ситуация вроде бы нормализовалась, но Майкл уже дрожал, как дикобраз среди воздушных шариков.

В 6:15 сайт в Нью-Йорке потерял связь с внешним миром. В PEER 1 не обнаружили никаких своих неисправностей. Майкл оделся, сел в метро и поехал на Манхэттен. Сервер работал. Соединение с сетью PEER 1 было нормальным. Проблема, видимо, заключалась в сетевом коммутаторе. Майкл отключил коммутатор, подключил наш роутер прямо к роутеру PEER 1 – и мы снова были в Интернете.

К утру, когда большинство наших американских клиентов пришли на работу, все было прекрасно. По электронной почте начали поступать жалобы от наших европейских клиентов. Проведя расследование, Майкл выяснил, что проблема возникла из-за особенностей настройки коммутатора. Этот коммутатор может работать на разных скоростях (10, 100 или 1000 мегабит в секунду). Можно задать скорость вручную либо позволить устройству автоматически установить самую высокую скорость, допустимую для нормальной работы двух связанных устройств. В отказавшем коммутаторе была задана автоматическая настройка. Она работает, но не всегда, и 10 января она не сработала.

Майкл знал о возможности этой проблемы, но при установке коммутатора забыл установить скорость, поэтому коммутатор остался с заводскими настройками, в которых задан режим автоматического выбора скорости, что не вызывало никаких проблем. До поры до времени.

Майкл расстроился. Он прислал мне электронное письмо:

Я знаю, что у нас нет официального SLA для хостинга On Demand, но было бы полезно составить его хотя бы для внутреннего упот-

ребления. С его помощью можно было бы оценить, в какой степени я или команда (будущего) системного администратора обеспечиваем общие задачи бизнеса. Я потихоньку над ним работаю, но в свете сегодняшних событий хочется ускорить этот процесс.

Обычно в SLA сказано о времени «бесперебойной работы», поэтому нужно определить, что понимается под «бесперебойной работой» в контексте On Demand. Выяснив это, можно задать политику, написать ряд скриптов для контроля/отчетности и периодически проверять, как мы «выполняем собственные обещания».

Отличная мысль!

Но с SLA связаны некоторые проблемы. Главное – нет осмысленной статистики, потому что отказов очень мало. Насколько я помню, с тех пор как полгода назад мы стали предоставлять хостинг FogBugz On Demand, было всего два незапланированных отключения, считая данное. Одно из них произошло по нашей вине. У большинства надежных сетевых сервисов бывает два, самое большее три отказа за год. Когда случаев отказа так мало, большое значение приобретает их продолжительность, а она может очень сильно варьировать. Внезапно все упирается в то, как скоро кто-то доберется до оборудования, чтобы устранить неисправность. Действительно высокий уровень бесперебойной работы не должен зависеть от того, кто заменяет неисправную деталь. Нельзя даже ждать, пока он определит, что сломалось: нужно заранее выявить все, что может выйти из строя, а это практически невозможно. Опасны неожиданные неожиданности, а не те, к которым вы подготовились.

Обеспечение действительно высокого коэффициента надежности обходится очень дорого. Пресловутые «шесть девяток» (99,9999% готовности к работе) соответствуют тридцати секундам простоя в год. Это просто смешно. Даже тот, кто заявляет, что построил какую-то ультрадорогую сверхизбыточную суперпуперсистему с шестью девятками, проснется в один прекрасный день – не знаю, когда, но он наступит непременно, – и обнаружит, что произошло нечто совершенно непредвиденное – ну, там, три электромагнитные бомбы, по одной в каждый центр обработки данных, – и хоть всю голову сломай, но две недели ничего работать не будет.

Думать надо вот о чем: если ваша система с шестью девятками по каким-то таинственным причинам хоть раз выйдет из строя, то даже выяснив и устранив причину за час, вы исчерпаете свой бюджет допустимых отка-

зов на целое столетие вперед. Даже у таких известных своей надежностью систем, как служба междугородней связи AT&T, случается длительный перебой в работе (6 часов в 1991 году), понижающий их уровень до малопочтенных трех девяток, а ведь междугородняя связь AT&T считается службой связи высшего класса, золотым стандартом надежности.

Для непрерывного обслуживания через Интернет характерна проблема *черного лебедя*. Нассим Тадлеб (Nassim Taleb), придумавший этот термин, определяет его так (*www.edge.org/3rd_culture/taleb04/taleb_indexx.html*): «Черный лебедь – это выброс, событие вне области обычных ожиданий». Почти все перебои в работе Интернета являются неожиданными неожиданностями – крайне маловероятными событиями за гранью ожидаемого. Такие вещи случаются настолько редко, что пользоваться обычными статистическими представлениями вроде «среднего времени между отказами» бессмысленно. Какое может быть «среднее время между катастрофическими наводнениями в Новом Орлеане»?

Длительность перебоев в течение года не позволяет предсказать, сколько минут перебоев будет в следующем году. Это напоминает нынешнее положение в коммерческой авиации: Совет по безопасности на транспорте потратил столько сил на искоренение стандартных причин аварий, что каждая авиакатастрофа, которую он теперь расследует, представляется безумным разовым выбросом – черным лебедем.

Где-то посреди между «крайне ненадежным» уровнем обслуживания, когда постоянно возникают дурацкие перебои, и «крайне надежным», когда миллионы вкладываются в то, чтобы рабочее время увеличилось всего на минуту за целый год, есть наилучшая область, в которой приняты меры против всех ожидаемых неожиданностей. Отказ одного жесткого диска допускается и не выводит вас из строя. Отказ одного сервера DNS допускается и не выводит вас из строя. Но неожиданные неожиданности могут привести к сбою. Это лучшее, на что можно рассчитывать.

Чтобы попасть в это милое место, мы воспользовались идеей Сакичи Тойода (Sakichi Toyoda), основателя компании Toyota. Он называет ее «пять почему». Когда что-то выходит из строя, снова и снова спрашивайте – почему, пока не выясните главную причину. Тогда вы сможете устранить эту причину, а не ее симптомы.

Поскольку это хорошо согласуется с нашей идеей двух способов, позволяющих все исправить, мы решили попробовать применить «пять почему» в своей практике. Вот что получилось у Майкла:

Мы потеряли соединение с PEER 1 NY.

- Почему? – Видимо, наш коммутатор перевел порт в нерабочее состояние.
- Почему? – После обсуждения с NOC PEER 1 мы пришли к предположению, что это могло быть вызвано несоответствием скорости/дуплекса Ethernet.
- Почему? – В интерфейсе коммутатора был задан автоматический выбор скорости вместо ручной настройки.
- Почему? – Мы знали о возможности подобной проблемы и прежде. Но у нас нет письменных стандартов и процедур проверки конфигурации промышленных коммутаторов.
- Почему? – Документация часто считается вспомогательным средством на случай, если поблизости нет системного администратора, или для другого технического персонала, тогда как в действительности она должна содержать перечень необходимых проверок.

«Если бы мы сначала написали стандарт, а потом установили коммутатор и проверяли свою работу на соответствие стандарту, этого перебоя не было бы, – писал Майкл. – Или, случись он однажды, в стандарт были бы внесены соответствующие изменения».

После некоторого обсуждения мы пришли к согласию, что нам не нужны бессмысленные для статистики измерения, а нужен процесс непрерывного совершенствования. Мы не станем заключать SLA со своими клиентами, а заведем блог, в котором будем в реальном времени регистрировать каждый перебой в работе, обеспечим полный разбор происшедшего, зададим «пять почему», доберемся до главной причины и сообщим своим клиентам о мерах, принятых для исключения возможности повторения инцидента в будущем. В данном случае принято решение включить во внутреннюю документацию подробные перечни действий для всех эксплуатационных процедур предоставления хостинга.

Из блога клиент может узнать, чем были вызваны проблемы и как мы стараемся улучшить обслуживание, что, надеемся, убедит его в неуклонном повышении качества.

В то же время, наша служба работы с клиентами имеет право кредитовать счета клиентов, считающих, что перебой нанес им ущерб. Мы даем возможность клиентам самим решать, какой размер возмещения в пределах месячной платы их удовлетворит, потому что не каждый клиент даже

заметит временную недоступность услуги, не говоря уже о том, что пострадает от этого. Надеюсь, что эта система повысит нашу надежность до такого уровня, когда единственный перебой, который у нас возникнет, окажется действительно черным лебедем.

P. S. Кстати, мы хотим взять еще одного системного администратора, чтобы будить посреди ночи не только Майкла.

ГЛАВА ТРИДЦАТЬ ШЕСТАЯ

Установите приоритеты

12 ОКТЯБРЯ 2005 ГОДА, СРЕДА

Пора было заканчивать возню с FogBugz 4.0 и начинать работать над 5.0. Мы только что выпустили большой пакет обновлений, исправив тьму мелких ошибок, которые никто никогда не заметил бы (и добавив пару новых мелких дефектов, которые тоже никто никогда не заметит), и пора было добавлять настоящие новые функции.

К тому времени, когда мы собрались начать разработку, идей было столько, что хватило бы для 1700 программистов на пару десятилетий. К сожалению, у нас было всего три программиста, а выпустить новую версию мы планировали следующей осенью, так что пришлось расставить некоторые приоритеты.

Прежде чем рассказывать о том, как мы упорядочили свой список функций, приведу два примера того, как это делать не нужно.

Случай первый: если вдруг окажется, что вы собрались реализовать функцию, обещанную одному клиенту, в вашем мозгу должна загореться КРАСНАЯ АВАРИЙНАЯ ЛАМПОЧКА. Если вы делаете что-то для одного клиента, это значит, что либо ваш агент по продажам совершенно вышел из под контроля, либо вы начали опасное скатывание в сторону *консалтингвера*[1] (consultingware). Ничего дурного в консалтингвере нет: скатывание

[1] По определению Джоэла, консалтингвер – разновидность коробочного продукта, для работ по настройке и установке которой требуется целая армия консультантов, что обходится недешево. – *Прим. перев.*

в него происходит достаточно комфортно, но такой продукт приносит меньше прибыли, чем коробочный.

Коробочный продукт – это модель разработки, при которой покупатель берет все или ничего. Разрабатываете программу, делаете для нее упаковочную коробку, и покупатель либо берет ее, либо нет. Вопрос о том, что клиент купит продукт, если в него будет добавлена еще одна функция, не ставится. Никто не приглашает вас, чтобы обсудить, какие функции должны быть в программе. Нельзя позвонить в Microsoft и сказать: «Послушайте, мне нравится в вашем Excel функция BAHTTEXT для записи чисел прописью на тайском языке, но я хотел бы воспользоваться такой же функцией для английского языка. Я куплю Excel, если вы реализуете там эту функцию». Потому что если вы звоните в Microsoft, то слышите следующий ответ:

«Спасибо, что обратились в Microsoft. Если у вас есть специальный четырехзначный рекламный код, нажмите 1. Для получения технической поддержки по всем продуктам Microsoft нажмите 2. Для предпродажного лицензирования Microsoft или получения информации о программе нажмите 3. Если вы знаете номер сотрудника Microsoft, с которым хотите переговорить, нажмите 4. Чтобы повторить это сообщение, нажмите звездочку».

Понятно? Нигде не сказано: «Для обсуждения того, какие функции нужно добавить в наши продукты, чтобы вы их купили, нажмите 5».

Разработка под клиента – темная история. Клиент говорит вам, что нужно сделать, а вы спрашиваете: «Вы уверены в этом?», и он подтверждает, и вы пишете совершенно замечательную спецификацию и спрашиваете: «Вы этого хотите?», и он подтверждает, и вы заставляете его подписать спецификацию несмываемыми чернилами (нет, *кровью!*), и он подписывает, и вы создаете то, под чем он подписался, – быстро и точно – и показываете клиенту, и он в ужасе и шоке, и до конца недели вы выясняете, включает ли в себя ваша страховка от ошибок и упущений покрытие судебных расходов в тяжбе, в которую вы ввязались, или только стоимость урегулирования претензий. Либо, если очень повезет, заказчик вяло улыбнется и уберет ваш код в ящик, чтобы никогда им не пользоваться и никогда больше вам не звонить.

Консалтингвер занимает некоторое промежуточное положение: вы якобы делаете коробочный продукт, хотя на самом деле – заказное программное обеспечение. Вот как это происходит:

1. Вы работаете по найму и пишете код для обувной фабрики, а

2. Фабрике нужна программа для наведения глянца на ботинки, поэтому
3. Вы пишете программу полировки ботинок на VB 3.0 с элементами JavaScript, Franz Lisp и базы данных FileMaker, которая работает на старом Маке, подключаемом к сети по AppleScript, после чего
4. Все говорят, что программа первый сорт, и поскольку вы всегда мечтали создать собственную софтверную компанию, а то и стать кем-то вроде Билла Гейтса или хотя бы Ларри Эллисона, то
5. Вы покупаете у своей прежней фирмы права на ShoeShiner 1.0 и находите венчурный капитал, чтобы основать собственную фирму ShoeShiner LLC, которая рекламирует ПО для полировки ботинок, но
6. Никто из бета-тестеров не может заставить ваш продукт работать из-за неясной зависимости от AppleScript и жестко прошитого в исходном коде IP-адреса, поэтому установка у каждого клиента занимает месяц, и
7. Трудно находить клиентов, потому что ваш продукт очень дорог из-за всех этих расходов на установку, включающих в себя древний Macintosh IIci под System 7, который приходится покупать на eBay в музеях компьютеров, так что ваши инвесторы начинают нервничать, пытаясь
8. Прижать агента по продажам,
9. Который обнаруживает, что одному из ваших потенциальных клиентов не нужно полировать ботинки, но вот программа для утюжки брюк ему действительно пригодилась бы, и
10. Агент по продажам – чего еще от него можно ожидать – продает программу для утюжки брюк за 100 K долларов,
11. И теперь вы полгода пишете этому клиенту уникальный «модуль для утюжки брюк»,
12. Который никогда не понадобится больше никому, поэтому, фактически,
13. Во всех практических смыслах тот год, что вы потратили на поиски инвесторов, можно было спокойно работать на жалованье в компании, шьющей брюки; `GOTO 1`.

Не спешите, лучше постарайтесь держаться той стороны, где делают коробочный продукт. У такого продукта нет маргинальных затрат на каждого нового покупателя, поэтому, в сущности, вы продаете одну и ту же вещь снова и снова, зарабатывая гораздо больше. Более того, вы можете снизить

цену, потому что делите издержки разработки на гораздо большее количество клиентов, а снизив цену, *получаете новых покупателей*, потому что многие вдруг осознают, что ваш подешевевший продукт достоин внимания, и жизнь прекрасна, и все чудесно.

Поэтому если вдруг окажется, что вы реализуете функцию только потому, что она обещана клиенту, значит вы сдвигаетесь в область консалтинг-вера и заказных разработок, что прекрасно, если вам это нравится, но там нет таких возможностей получить прибыль, как с коммерческими коробочными продуктами.

Это не означает, что не нужно прислушиваться к своим клиентам. Я, например, считаю, что пора бы Microsoft действительно реализовать версию функции `ВАНТТЕХТ` для тех из нас, кто еще не приобщился к мировой экономике и не выучил тайский, по старинке выписывая чеки в других валютах. На самом деле, если вам очень хочется думать, что лучший способ применения ресурсов разработки – позволить своим крупнейшим клиентам делать «заявки» на функции, что ж, можно поступить и так, но вскоре вы убедитесь, что крупным богатым клиентам требуются функции несколько иного рода, чем массовому рынку, и что новая функция для работы с такой денежной единицей, как бат, мало помогает повысить продажи Excel для курортов в Скоттсдейле (Аризона), и в реальности вы лишь отдаете разработчиков в кабалу агентам по продаже, единственная цель которых – увеличить свои комиссионные.

Идя таким путем, Биллом Гейтсом не стать.

А теперь я расскажу о втором пути, по которому не надо идти, принимая решение о том, какие функции следует реализовать. Не делайте чего-то только потому, что это неизбежно. Неизбежность – планка недостаточной высоты. Сейчас поясню.

Как-то на первом году жизни Fog Creek я раскладывал какие-то документы и обнаружил, что у меня кончились синие папки.

Дело в том, что у меня есть система. Синие папки относятся к клиентам. Папки цвета манильской пеньки – для служащих. Красные папки – денежные поступления. Все остальное желтого цвета. Мне нужна была синяя папка, а они кончились.

И я сказал себе: «Все равно мне понадобятся синие папки, так что пойду сейчас в Staples и куплю их».

Разумеется, это было пустой тратой времени.

Размышляя об этом позднее, я понял, что уже давно занимаюсь всяким dumb shit (это технический термин) только потому, что считал: раз все равно придется этим заниматься, то можно сделать это прямо сейчас.

Под этим предлогом я пропалывал сад, заделывал дырки в стенах, сортировал диски MSDN (по цветам, языкам и номерам) и так далее, вместо того чтобы писать или продавать код – единственное, чем действительно стоит заниматься в начинающей компании.

Иными словами, я просто делал вид, что все задачи, которые надо было решить, имеют одинаковую важность, а потому, раз они все равно неизбежны, то можно решать их в любом порядке! Во как!

Честно говоря, я просто откладывал дела на потом.

Что я должен был сделать в действительности? Для начала, следовало избавиться от фетишизма в отношении цвета папок. Нет никакой разницы. Совсем не обязательно иметь для папок цветовой код.

А что с этими дисками MSDN? Да бросить их в большую коробку. Идеально.

Но главное, я должен был раньше понять, что «важность» – величина не двоичная, а аналоговая. Есть самые разные степени важности, и если вы будете пытаться делать все, вы никогда не сделаете ничего.

Поэтому если хотите добиться успеха, в каждый данный момент времени вы должны безусловно знать, какая задача *сейчас* главная, и если вы занимаетесь чем-то другим, значит, двигаетесь вперед не самым быстрым из возможных способов.

Медленно, но я отвыкаю от привычки медлить. Я оставляю менее важные дела на потом. Одна милая дама из страховой компании надоедала мне целых два месяца, требуя данные для возобновления нашего полиса, а я не давал ей эти данные, пока она не потребовала их в пятидесятый раз, строго предупредив, что наша страховка закончится через три дня. Думаю, это правильно. Я начал подозревать, что если у вас на столе полный порядок, это может свидетельствовать о том, что вы недостаточно эффективно работаете.

Совершенно убийственная мысль!

Итак, не нужно реализовывать функции только потому, что ваш агент по продажам клятвенно обещал какому-то одному из ваших клиентов, что они будут сделаны, и не нужно реализовывать первыми маловажные/развлекательные функции только потому, что «все равно в итоге придется их делать».

Вернемся все же к нашей теме – выбору функций для FogBugz 5.0. Вот как мы впервые упорядочили их.

Сначала я взял пачку карточек размером 5×8, записав на каждую по одной функции. Затем я собрал всю команду. Мой опыт показывает, что такой способ действует, когда собирается до двадцати человек, причем полезно пригласить тех, чьи точки зрения на все различны: программистов, проектантов, тех, кто общается с клиентами, агентов по продажам, менеджеров, составителей документации, тестеров и даже (!) клиентов.

Я также попросил всех прийти на совещание со своим личным списком идей относительно будущих функций. В первой части совещания мы вкратце прошлись по всем функциям, с тем чтобы у всех было примерно одинаковое общее представление о каждой из функций и чтобы для каждой функции была заготовлена карточка.

Идея была в том, чтобы на этом этапе не спорить о достоинствах каждой функции, и не проектировать ее, и даже не обсуждать, а просто составить приблизительное представление о ее сути. Вот некоторые из функций, предлагавшихся для FogBugz 5.0:

- Персонализация исходной страницы
- Удобное составление графиков
- Учет оплачиваемого времени
- Ветвление ошибок
- (Еще 64 функции...)

Весьма туманно. Напоминаю, что в тот момент нас не интересовало, как должна быть реализована каждая функция или что для нее требовалось, потому что нашей целью была грубая расстановка приоритетов, на основе которой можно начинать разработку. В результате мы составили список примерно из 50 крупных функций.

Во второй части мы прошлись по всем функциям, и каждый из присутствующих проголосовал по каждой функции: просто «за» или «против». Никакого обсуждения, ничего больше: большой палец вверх или большой палец вниз по каждой функции. В результате выяснилось, что четырнадцать функций собрали очень слабую поддержку. Я выкинул все функции, получившие один-два голоса, оставив тридцать шесть потенциальных функций.

Затем мы назначили «цену» каждой функции по десятибалльной шкале: быстро реализуемая функция получала 1, а функция-монстр – 10. Здесь нужно помнить, что задачей было не планирование разработки функций,

а их разделение на группы – мелкие, средние и крупные. Я поочередно называл функции, предлагая разработчикам оценить каждую как «маленькую», «среднюю» или «большую». Даже не зная, сколько времени займет реализация функции, легко прикинуть, что «ветвление ошибок» – «маленькая» функция, а нечто туманное вроде «персонализации исходной страницы» – «большая». Исходя из общих оценок и моих собственных представлений, мы определили стоимость всех функций:

Функция	Цена (долларов)
Персонализация исходной страницы	10
Удобное составление графиков	4
Учет оплачиваемого времени	5
Ветвление ошибок	1

Снова подчеркну, что все это довольно нестройно, неточно и не важно. Вы же не составляете сейчас график работы – вы просто расставляете приоритеты. Нужно только прикинуть, что за один и тот же промежуток времени можно сделать две средние функции, или одну большую, или десять маленьких. Большая точность здесь не нужна.

На следующем этапе мы составили меню из всех тридцати шести предложенных функций и их цен. Каждый из участников получил один экземпляр меню и 50 долларов на расходы. Можно было как угодно потратить деньги в пределах 50 долларов. Разрешалось купить половинку функции или двойную функцию. Тот, кому очень понравился «учет оплачиваемого времени», мог потратить на эту функцию 10 или 15 долларов; тот, кому она нравилась лишь отчасти, мог потратить на нее 1 доллар в расчете, что ее профинансируют другие.

Затем мы просуммировали, сколько потрачено на каждую функцию:

Функция	Цена	Потрачено
	(долларов)	
Персонализация исходной страницы	10	12
Удобное составление графиков	4	6
Учет оплачиваемого времени	5	5
Ветвление ошибок	1	3

Наконец, я поделил истраченную сумму на цену:

Функция	Цена (долларов)	Потрачено (долларов)	
Персонализация исходной страницы	10	12	1,2
Удобное составление графиков	4	6	1,5
Учет оплачиваемого времени	5	5	1,0
Ветвление ошибок	1	3	3,0

А потом отсортировали функции по полученному коэффициенту, чтобы определить самые популярные:

Функция	Цена (долларов)	Потрачено (долларов)	
Ветвление ошибок	1	3	3,0
Удобное составление графиков	4	6	1,5
Персонализация исходной страницы	10	12	1,2
Учет оплачиваемого времени	5	5	1,0

Готово! Вот список всех функций, которые можно реализовать, примерно в порядке общих представлений о том, какие из функций наиболее важны.

А теперь можно заняться уточнением. Можно связывать между собой функции, естественным образом тяготеющие друг к другу, например, составление графиков облегчает учет времени, поэтому имеет смысл реализовать обе функции вместе или ни одной. А иногда, глядя на упорядоченный список, вы видите, что в нем явно что-то перепутано. Так переставьте местами! Здесь ничто не высечено на камне на века. Менять приоритетность можно даже во время разработки.

Но больше всего меня поразило то, что в итоге мы получили список, который очень хорошо устанавливал приоритеты для FogBugz 5.0 и действительно отражал общее согласие в отношении сравнительных приоритетов разных функций.

Мы начали реализацию функций примерно в порядке списка приоритетов, чтобы завершить ее к марту, чтобы, покончив с добавлением новых

функций, перейти к этапу интеграции и тестирования. Для каждой (неочевидной) функции перед реализацией составляется спецификация.

(Ворчливые счетчики очков в соревновании BDUF/Agile[1] должны быть окончательно сбиты с толку. «Он за кого – за BDUF? Или за Agile? Чего он хочет? *Может он хоть раз* определить свою позицию?»)

Весь процесс планирования занял три часа.

Если вам повезло и у вас есть возможность выпускать новые версии чаще, чем это удается нам (см. главу 34), все равно нужно действовать в соответствии с упорядоченным списком – просто вы можете чаще останавливаться и выпускать новую версию. Частый выпуск версий хорош тем, что можно регулярно переупорядочивать список, основываясь на реакции пользователей, но такую роскошь допускает не каждый продукт.

Майк Конте (Mike Conte) научил меня этой системе во время планирования выпуска Excel 5, когда мы справились с задачей за пару часов, собрав примерно два десятка людей в конференц-зале. Самое замечательное, что примерно 50% функций, на которые у нас не хватало времени, в действительности оказались глупыми, и без них Excel только выиграл.

Система не идеальна, но это лучше, чем ходить в Staples за синими папками. Вот так-то.

[1] BDUF – детальное проектирование, Agile – ускоренное программирование. – *Прим. перев.*

Алфавитный указатель