БИБЛИОТЕКА ПРОГРАММИСТА

Джоэл Х. Спольски

Лучшие примеры разработки ПО

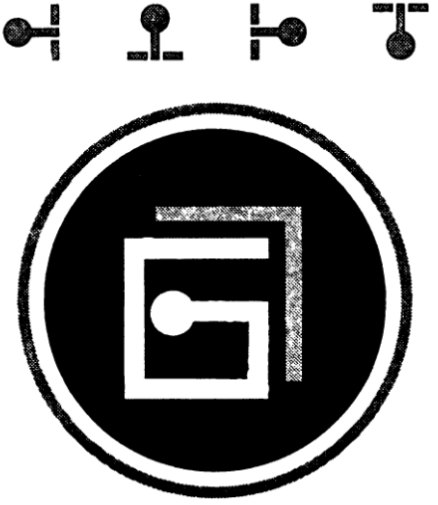
Эта книга является сборником статей, эссе, заметок и просто умных мыслей программистов, разработчиков и пользователей программного обеспечения на тему, такую же древнюю, как и сами компьютеры:

- **Как надо писать программы?**
- **Как не надо писать программы?**

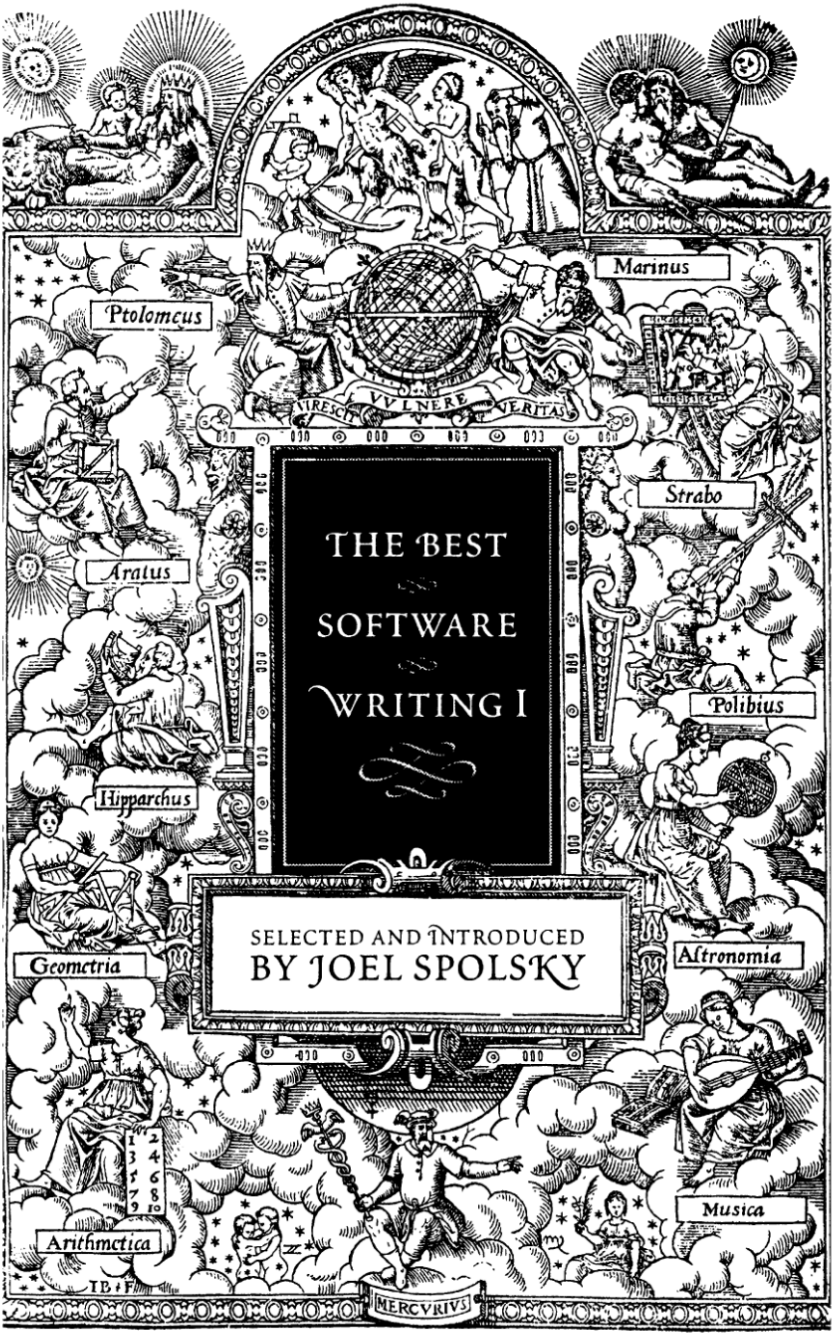
Автор создал восхитительный коктейль из суждений и концепций, приправленный изысканным стилем настоящего эксперта.

a! Apress

ПИТЕР



ПИТЕР®



THE BEST SOFTWARE WRITING I

SELECTED AND INTRODUCED BY JOEL SPOLSKY

БИБЛИОТЕКА ПРОГРАММИСТА

Джоэл Х. Спольски

Лучшие примеры разработки ПО

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2007

Джоэл Х.Спольски

Лучшие примеры разработки ПО

Перевел с английского Е. Матвеев

Заведующий редакцией	*А. Кривцов*
Ведущий редактор	*А. Пасечник*
Научный редактор	*Е. Матвеев*
Технический редактор	*Л. Родионова*
Литературный редактор	*А. Пасечник*
Художник	*Л. Адуевская*
Корректор	*И. Смирнова*
Верстка	*Л. Родионова*

ББК 32.973-018
УДК 004.413

Спольски Дж. Х.
С73 Лучшие примеры разработки ПО. — СПб.: Питер, 2007. — 208 с.: ил.

ISBN 5-469-01291-3

Перед вами книга Джоэла Спольски — ветерана индустрии программного обеспечения. Его электронный журнал «Joel on Software» (http://www.joelonsoftware.com) стал одним из самых популярных независимых веб-изданий среди программистов.

Эта книга — не учебник, не документация, не набор методик и (или практик). Это классное чтиво для разработчика со стажем и мозгами. Это иллюстрации по поводу того, как можно вообще относиться ко всему, что делаешь и что делается вокруг тебя. Это, в конце концов, набор сумасшедших идеек, которые могут пригодиться и в жизни.

ISBN 5-469-01291-3

ISBN 1590595009 (англ.)

ООО «Питер Пресс», 198206, Санкт-Петербург, Петергофское шоссе, д. 73, лит. А29.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.
Подписано в печать 28.09.06. Формат 70×100/16. Усл. п. л. 13. Тираж 2000. Заказ 777
Отпечатано с готовых диапозитивов в ОАО «Техническая книга».
190005, Санкт-Петербург, Измайловский пр., д. 29.

Содержание

На обложку книги был вынесен титульный лист первого полного англоязычного издания «Начал» Евклида, опубликованного в 1571 году.

Интересно заметить, что эту книгу перевел с греческого сэр Генри Биллингсли, ставший лорд-мэром Лондона (вряд ли кто-нибудь из современных политиков обладает познаниями в математике и греческом, необходимыми для выполнения такой работы). В переводе Биллингсли помогал Джон Ди — тот самый Джон Ди, прославленный в художественной литературе (например, в рассказе Г. Лавкрафта «Ужас Данвича»). Кстати говоря, большинство ученых того времени, включая самого Ньютона, принадлежали к обоим мирам, науки и магии. Джон Ди написал замечательное к книге предисловие, в котором утверждалась центральная роль математики во всех науках и искусствах — тем самым он завоевал расположение всех математиков последующих поколений, включая нашего издателя Гэри Корнелла (Gary Cornell). В частности, в предисловии говорилось:

«Сколь великий соблазн, сколь восхитительное обольщение — наука, главная тема которой так стара, так чиста, так превосходна; которая окружает всех живых существ, и так часто используется во всемогущей и непостижимой мудрости Творца».

При всей своей важности, перевод не был лишен недостатков. Самый заметный из них — ошибка на титульном листе (показанном на рисунке): Биллингсли приписал «Начала» Евклиду из Мегары. Но в действительности Евклид Александрийский работал в великой библиотеке и написал «Начала»!

Наша копия титульного листа позаимствована из экземпляра, хранящегося в библиотеке Бэнкрофта Калифорнийского университета в Беркли. Она была отсканирована с разрешением 2400 dpi хорошими людьми из библиотеки Бэнкрофта. Из изображения был удален красновато-бурый оттенок, характерный для старых книг, а также внесены другие необходимые усовершенствования; этим занимался Курт Крамс, главный дизайнер Apress, в Adobe Photoshop на Mac G4.

О редакторе

Джоэл Спольски — эксперт в области разработки программного обеспечения с мировым именем. Его сайт Joel on Software (www.joelonsoftware.com) пользуется популярностью среди разработчиков со всего мира и был переведен более чем на 30 языков. В качестве основателя компании Fog Creek Software из Нью-Йорка он создал популярную систему управления проектами FogBugz. Джоэл работал на Microsoft, где участвовал в проектировании VBA в качестве участника команды Excel, и в Juno Online Services, где он разрабатывал Интернет-клиента, используемого миллионами людей. Он является автором двух книг: «User Interface Design for Programmers» (Apress, 2001) и «Joel on Software» (Apress, 2004). Джоэл обладает степенью бакалавра Йельского университета по информационным технологиям. До поступления в университет от служил в Вооруженных силах Израиля десантником и был одним из основателей киббуца Ханатон.

Об авторах

Кен Арнольд (Ken Arnold) долгие годы слонялся без определенных занятий в компьютерных областях; в частности, он посещал Беркли во время работы над проектом BSD, создавал библиотеку curses и помогал в работе над rogue; вел рубрику «The C Advisor» для Unix Review (позднее «The C++ Advisor»); был соавтором «The Java Programming Language» (русский перевод: «Язык программирования Java», Питер, 1997) и других книг; проектировал JavaSpaces и участвовал в проектировании Jini; время от времени прикидывался модным, ведя блог. Его текущие увлечения — человеческие факторы языков программирования и API, надежные системы электронного голосования и подключаемое оформление Napkin для Java.

Леон Бамбрик (Leon Bambrick) — плодовитый программист, сатирик и спорщик, проживающий в южном полушарии. Он впервые встретился с Джоэлом Спольски, когда они вместе оказались на необитаемом острове, где у них не было ничего, кроме 8086 и экземпляра книги Кернигана–Ричи. Его веб-сайт secretGeek.net упоминается в 3 эпизоде «Звездных Войн».

Майкл Бин (Michael Bean) — разработчик и бизнесмен. В настоящее время является президентом и одним из основателей Forio Business Simulations. До работы в Forio Майкл занимал высшие руководящие должности в фирмах, занимавшихся консультациями и разработкой, в Соединенных Штатах и Европе. Майкл также был научным сотрудником

в группе динамики систем в MIT, где разрабатывал имитации для анализа стратегических последствий решений, принимаемых управленцами. Майкл консультировал корпорации и правительственные агентства (как национальные, так и зарубежные) по вопросам ценообразования, стратегии конкуренции, развивающимся технологиям и клиентской миграции. Он проводил семинары по планированию сценариев, системному мышлению и компьютерным имитациям. Также Майкл читал доклады на национальных конференциях по стратегиям, программному обеспечению и компьютерным имитациям.

Рори Блайс (Rory Blyth) работает жалкой корпоративной марионеткой в Microsoft. В свое свободное время он ведет блог по адресу www.neopoleon.com, размышляет над смыслом жизни и считает себя одним из трех пропавших камней Шанкары. Наверное, он таковым не является, но это помогает ему избавиться от комплекса неполноценности.

Адам Босуорт (Adam Bosworth) недавно присоединился к Google в качестве вице-президента по техническим вопросам. Он пришел в Google из BEA, где был главным архитектором и вице-президентом по современным разработкам, а также отвечал за координацию технической работы в отделе инфраструктур. До прихода в BEA он основал Crossgain — компанию по разработке программного обеспечения, приобретенную BEA. Известен как один из пионеров XML, занимал различные руководящие посты в Microsoft, в том числе пост руководителя по общим вопросам в WebData — группе, занимавшейся определением и развитием стратегии XML. Во время работы в Microsoft отвечал за проектирование и выпуск Microsoft Access (база данных для PC), а также за формирование и руководство командой разработчиков HTML-ядра для Internet Explorer 4.0.

Дана Бойд (Danah Boyd) — докторант в Школе информационного менеджмента и систем в Калифорнийском университете, Беркли. Здесь она изучает передачу людьми собственного представления в социальных контекстах неизвестной аудитории этнографическими методами. Она особенно интересуется развитием молодежи и культурным самопознанием и роли технологий в этом процессе. До поступления в Беркли Дана получила степень магистра в области социальных сред в MIT и степень бакалавра в области информационных технологий в университете Брауна. Дана часто пишет в блогах в Apophenia (www.zephoria.org/thoughts) и Many-to-Many (www.corante.com/many).

Рэймонд Чен (Raymond Chen) работал в Microsoft с 1992 года и многое повидал. Его блог посвящен истории Windows и умирающему искусству программирования для Win32.

Кевин Ченг (Kevin Cheng) — независимый специалист в области пользовательских интерфейсов и космополит. Обладает ученой степенью магистра в области взаимодействий «человек-компьютер» и эргономики в UCLIC (University College London Interaction Centre). Является одним из основателей и создателей OK/Cancel (www.ok-cancel.com) — участника пятерки лучших сайтов, посвященных практичности и взаимодействиям человека с компьютером.

Том Чи (Tom Chi) обладает степенью магистра в области электротехники. Наверное, из этого должно следовать, что его квалификация не позволяет ни рассуждать на темы взаимодействий человека с компьютером, ни писать всякие странные вещи. Тем не менее, неделю за неделей он продолжает предаваться несбыточным мечтаниям на ok-cancel.com. Что касается заслуг, он может похвастаться проектированием интерфейса для двух версий Outlook и темной историей с консультированием клиентов, входящих в «список 500» из журнала Форбс — но приличные люди на такие темы не говорят. Тссс.

Кори Доктороу (Cory Doctorow) (craphound.com), координатор по делам Европы в EFF (Electronic Frontier Foundation, www.eff.org) — некоммерческой организации, работающей над обеспечением гражданских свобод в технологическом законодательстве, политике и стандартах. Он представляет интересы EFF в различных комитетах по стандартизации и консор-

циумах, а также во Всемирной организации по охране интеллектуальной собственности (WIPO, World Intellectual Property Organization). Доктороу является автором многочисленных публикаций в журналах Wired, Make, and Popular Science, а его научно-фантастические работы получали премии Campbell, Sunburst и Locus. Участвует в редактировании популярного блога Boing Boing (boingboing.net). Родился в Канаде, сейчас проживает в Лондоне.

Брюс Эккель (Bruce Eckel, www.BruceEckel.com), является автором книг «Thinking in Java» (Prentice Hall, 1998, 2nd edition, 2000, 3rd edition, 2003, 4th edition, 2005), компакт-диска «Hands-On Java Seminar» (доступен на сайте), «Thinking in C++» (PH 1995; 2nd edition 2000, том 2 в соавторстве с Чаком Эллисоном, 2003) и «C++ Inside & Out» (Osborne/McGraw-Hill, 1993), а также других работ. Провел сотни презентаций по всему миру, опубликовал более 150 статей в различных журналах, был одним из основателей комитета ANSI/ISO C++, регулярно выступает на конференциях. Проводит общедоступные и закрытые семинары, консультирует в области разработок на C++ и Java.

Пол Форд (Paul Ford), редактор журнала Harper's, часто выступает с комментариями в программе NPR «All Things Considered» и является единственным владельцем сайта Ftrain.com. Возился с компьютерами последние 20 лет и не собирается прекращать. Живет в Бруклине, Нью-Йорк.

Пол Грэхем (Paul Graham) — автор очерков, программист и проектировщик языков программирования. В 1995 году вместе с Робертом Моррисом разрабатывал первое веб-приложение Viaweb, которое было приобретено Yahoo в 1998 году. В 2002 году описал простой байесовский спам-фильтр, ставший основой для большинства современных фильтров. В настоящее время работает над новым языком программирования Arc и новой книгой (вероятно, для издательства O'Reilly); является одним из партнеров Y Combinator. Пол является автором книг «On Lisp» (Prentice Hall, 1993), «ANSI Common Lisp» (Prentice Hall, 1995) и «Hackers & Painters» (O'Reilly, 2004). Обладает степенью бакалавра гуманитарных наук Корнельского университета и докторской степенью в области вычислительной техники в Гарварде. Изучал живопись в RISD и Accademia di Belle Arti во Флоренции.

Джон Грубер (John Gruber) — независимый автор, веб-разработчик, проектировщик и фанат Mac. Сочетает все эти интересы на своем сайте Daring Fireball (http://daringfireball.net). Джон живет в Филадельфии с женой и сыном.

Грегор Хопе (Gregor Hohpe) руководит отделом интеграции в Thought Works, Inc. — специализированной фирме, занимающейся разработкой приложений и интеграционного сервиса. Грегор является признанным специалистом в области асинхронных архитектур на базе сообщений, а также соавтором книги «Enterprise Integration Patterns» (Addison–Wesley, 2004). Регулярно выступает на технических конференциях по всему миру и ведет сайт www.eaipatterns.com.

Рон Джеффрис (Ron Jeffries) начал заниматься программированием еще до рождения многих читателей. Он является обладателем ученых степеней по математике и вычислительной технике. Группы, в которых он участвовал, занимались построением операционных систем, компиляторов, реляционных баз данных и других приложений. Созданные Роном продукты принесли свыше полумиллиарда долларов прибыли, и теперь ему интересно, почему из этих денег ему ничего не досталось.

Эрик Джонсон (Eric Johnson) закончил Иллинойский университет со степенью бакалавра по вычислительной технике в 1993 году, и с тех пор работает в FactSet Research Systems. В настоящее время является директором по рыночным данным; живет с женой и двумя детьми на юго-западе Коннектикута. С ним можно связаться по адресу johnson.eric@gmail.com.

Эрик Липперт (Eric Lippert) работает разработчиком программного обеспечения в Microsoft с 1996 года. Провел первые пять лет в компании за работой над VBScript, JScript, Windows Script Host и другими сценарными технологиями; в последнее время в основном работал над Visual Studio Tools For Office. Ведет блог, где делится советами по поводу сценарных языков, безопасности и (изредка) романтических отношений. Когда Эрик не занимается программированием и не пишет о программах, он играет старые песни на старых пианино, пытается выпрямить мачту своей крошечной яхты, делает воздушных змеев или уговаривает друзей помочь ему с ремонтом его 97-летнего дома.

Майкл «Рэндс» Лопп (Michael «Rands» Lopp) работает техническим руководителем по разработке программного обеспечения в Кремниевой долине. В прошлом он работал во многих известных фирмах, включая Borland International, Netscape Communications, Apple Computer, а также в мелких компаниях, о которых вы, к сожалению, никогда не слышали. В свободное время ведет блог по адресу www.randsinrepose.com, где пытается оптимистически истолковать тот факт, что мир становится все теснее и теснее.

Ларри Остерман (Larry Osterman) работает на Microsoft с 1984 года. В те времена он трудился рядовым программистом над различными продуктами Microsoft®, включая MS-DOS, MS-NET, LAN Manager, Windows NT, Exchange и eHome; сейчас работает в группе Windows Multimedia Technologies. Ларри живет к северу от Сиэттла с женой Вэлори, двумя детьми, четырьмя кошками и двумя лошадьми.

Мэри Поппендик (Mary Poppendieck) — настоящий ветеран в области руководства и разработки продуктов с более чем 25-летним опытом работы в сфере информационных технологий. В настоящее время является президентом компании Poppendieck LLC в Миннесоте. Ее книга «Lean Software Development: An Agile Toolkit» получила премию Software Development Productivity Award в 2004 году.

Рик Шаут (Rick Schaut) рос в Грин-Бэй и Милуоки, штат Висконсин. В детстве сильно увлекался бейсболом. После окончания школы изучал экономику и вычислительную технику в Университете штата Висконсин в Милуоки. Рик поступил в Microsoft в 1990 году, и с тех пор работает над версиями Microsoft Word.

Клей Ширки (Clay Shirky) преподает в программе интерактивных телекоммуникаций Нью-Йоркского университета и работает с разными клиентами, включая Библиотеку Конгресса, Connecting for Health и Nokia, в области сетевых архитектур. Он пишет о культурных и экономических аспектах Интернета (архив можно найти по адресу shirky.com).

Эрик Синк (Eric Sink) — основатель фирмы SourceGear, независимого производителя инструментария для программистов. Другие статьи Эрика можно найти в блоге по адресу software.ericsink.com. Эрик живет в центральном Иллинойсе с женой, двумя дочерьми и одной старой кошкой.

Аарон Шварц (Aaron Swartz) — юный писатель, хакер и активист. Ранее был советником по метаданным в Creative Commons и членом группы RDF Core Working Group при W3C; в настоящее время учится в Стэнфордском университете, ведет популярный блог и начинает работать в одной из вновь созданных фирм.

why the lucky stiff — программист и автор, который не может похвастаться сколько-нибудь реальными достижениями. Если не считать того случая, когда он ногой развалил надвое целое здание.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства http://www.piter.com вы найдете подробную информацию о наших книгах.

Посвящается моей сестре Рут, כִּי יוֹדֵעַ כָּל-שַׁעַר עַמִּי, כִּי אֵשֶׁת חַיִל אָתְּ

Кен Арнольд

Стиль есть содержание

Python стал весьма интересным явлением: это был первый серьезный язык программирования со времен FORTRAN на перфокартах, в котором пропуски (whitespace) играли важную роль. В Python блоки определяются не заключением фрагментов кода в пары begin/end или { }, а простыми отступами. Вот и все.

Многие программисты инстинктивно скривились. «Компилятор должен игнорировать пропуски!» — заявили они, совершенно забывая, почему. Главная причина состоит в том, что пропуски обычно остаются невидимыми глазу... на то они и пропуски. Так, например, у стандартной утилиты Unix make некоторые строки должны начинаться с символа табуляции; если ваш редактор заменяет символ табуляции восемью пробелами (как любезно с его стороны!), make-файл неожиданно перестает работать без каких-либо объяснений. Так мы все запомнили: пропуски должны игнорироваться.

Такие дела.

Возможно, мы зашли слишком далеко.

И в этом проявляется красота Python.

В C-подобных языках (C, C++, Java) человеческий глаз воспринимает отступы как определение блока, но компилятор видит { и }. Следовательно, в тех случаях, где отступы не согласуются со скобками, предпочтение отдается тому, что менее заметно человеку — фигурным скобкам. Но зачем нужны два способа определения блоков, для человека и для компилятора? Почему нельзя придерживаться одного способа, чтобы внешний вид программы всегда соответствовал ее функциональности?

Кен Арнольд довел этот принцип, позаимствованный из Python, до логического завершения. Он предлагает нечто гораздо более радикальное — и его идея, как и многие великие идеи, настолько безумна, что вполне может сработать. — *Ред.*

...И тогда я решил заняться стилем программирования — аналогом реслинга на TV в языках программирования...

Наверняка я не раз пожалею об этом, и все же публично признаюсь: я — еретик. (В этом конкретном случае речь идет лишь о ереси в области проектирования языков программирования. Прочие признания подождут до другого раза).

Скажу прямо: практически в любом зрелом языке (C, Java, C++, Python, Lisp, Ada, FORTRAN, Smalltalk, sh, JavaScript и т. д.) все проблемы стиля программирования практически решены, и на них можно не обращать внимания. Но чтобы не возиться с ними в будущем, придется немало повозиться изначально: чтобы перейти от текущего положения дел к такому положению, при котором мы перестаем беспокоиться о стиле, следует внедрить его на уровне языка. Угу. Вы не ослышались. Я говорю, например, что следующее обновление стандарта ANSI C должно определить стиль C-программирования K&R[1] на уровне грамматики языка. Программы, использующие новые возможности, либо оформляются в стиле K&R, либо отвергаются компилятором как синтаксически недопустимые.

Здесь стоит сделать паузу. Когда я завел речь на эту тему в списке рассылки, мне пришлось повторить несколько раз. Народ не сразу понял, потому что не мог поверить, что кто-то говорит нечто подобное. А я именно это имею в виду. Например, я хочу, чтобы в следующей грамматике C любое ключевое слово обязательно отделялось от открывающей круглой скобки пробелом: конструкция `if (foo)` должна быть разрешенной, а `if(foo)` — нет. Не предупреждение, не необязательная проверка, а реальный запрет. Конструкция без пробела попросту недопустима. Она не компилируется.

А вот логическое обоснование в простейшей форме:

- **Предпосылка 1: для любого языка существует один распространенный стиль программирования, или число таких стилей невелико.**

  Как правило, стиль задается основоположником или первым автором документации, но со временем появляются и другие стили. Но даже в C стандартные стили можно пересчитать по пальцам (если не считать тривиальные разновидности).

- **Предпосылка 2: не существует и никогда не будет существовать стиля программирования, который бы обладал очевидными преимуществами по сравнению с любым из стандартных стилей.**

  Взгляните правде в глаза. Открыть новый стиль, который бы повышал производительность работы более чем на несколько процентов по сравнению со стандартными стилями, также маловероятно, как изобрести новую позицию в сексе (космонавты не в счет — пока сам не увижу, все равно не поверю).

- **Предпосылка 3: на разновидности стиля программирования тратится невесть сколько времени.**

  Задумайтесь: сколько проектов переформатирования/красивой печати можно найти только на SourceForge3[2]? Сколько разных настроек содержит любая конкретная IDE (в том числе и emacs) для форматирования кода? Сколько времени тратится на выбор стиля, его документирование, соблюдение и обновление? Сколько журналов CVS, ClearCase и т. д. забивается мусо-

[1] По инициалам Брайана Кернигана (Brian Kernighan) и Денниса Ритчи (Dennis Ritchie), авторов книги «C Programming Language» (Prentice Hall, 1988) — основополагающего труда в области программирования на C.

[2] См. http://sf.net.

ром из-за изменений формата? Сколько умственных усилий затрачено на споры по этой теме?

- **Предпосылка 4: в любом нетривиальном проекте желательно применять стандартный стиль программирования.**

  Вероятно, здесь особых возражений не будет. Степень жесткости требований зависит от проекта, но если над одним кодом начнут работать несколько участников с конфликтными стилями программирования, это создаст куда больше проблем, чем любой конкретный стиль создает для одного программиста. Во всех известных мне проектах имеется принятый стиль, даже если он и не оговаривается явно.

- **Заключение: если рассматривать весь программный код в мире как единый «проект» с единым стилем, работа станет более эффективной, чем при разрешенных отклонениях в стиле.**

Подумайте хорошенько. Все примеры программ оформляются одним стилем. Веб-страницы, журналы, газеты, электронная почта — один стиль. Исчезают проблемы переформатирования. Стихают споры на тему того, чей стиль лучше. Переформатирование становится историческим курьезом.

А самое главное: Не Будет Больше Стилевых Войн! Серьезно! Прикиньте, сколько попусту потраченного времени можно было бы потратить на что-нибудь более полезное, типа флейма «vi против emacs»! Или борьбу за мир во всем мире! Или изобретение классного рецепта шоколадного печенья! Выбирайте сами!

Конечно, глобальное введение единого стиля возможно только в том случае, если у людей буквально не останется выбора. Сколько программистов C заменяет ключевое слово while на «during», потому что оно лучше соответствует их представлениям о стиле? (Лиц, злоупотребляющих препроцессором, просим не беспокоиться. Хотя ладно, отзовитесь — мы вас запишем в свою программу по евгенике). А сколько опускает круглые скобки вокруг условия if? Они этого не делают, потому что не могут. А если могли — наверняка кто-нибудь попытался бы. Все эти «персональные стили» останавливает лишь то, что компилятор C их не принимает. Если код не компилируется, его нужно исправить. Все до смешного просто... Поэтому и работает.

Поэтому я хочу, чтобы владельцы языковых стандартов взялись за дело. Я хочу, чтобы следующие версии этих языков потребовали, чтобы любой код, использующий новые возможности, соответствовал некоторому стилю. Пусть комитеты по стандартам скрежещут зубами, ноют и заламывают руки, наблюдая за тем, какой из стандартных стилей станет победителем. Продавайте билеты. Мы все выскажемся, а умные головы разработчиков языковых стандартов пускай решают. Понятно, чем все кончится — C отправится по пути K&R; C++ выберет стиль Бьярна (от которого я, уж простите, не в восторге); для Java будет выбран стиль Sun, представленный в спецификации языка и большинстве книг Java от Sun (включая мою); стиль Lisp почти что выбит в камне. Perl представляет собой необъятную трясину лексических и синтаксических помоев; никто из программистов не умеет толком отформатировать даже свой собственный код, но зато это единственный из известных мне крупных языков, у которого нет даже одного приличного стиля (вероятно, не считая совсем новый, но очень похожий на Java язык C#).

Некоторые вещи либо не проверяются в принципе (венгерская запись, префиксы методов get и set), либо еще не стали общепринятыми (скажем, порядок import/#include). Их можно оставить для будущих стандартов. А можно и не оставлять. Пускай владельцы стандартов решают. Но что бы они ни решили, пусть установят единый стиль и встроят его в эту чертову грамматику. Глобальная ересь включает одну серьезную ересь второго уровня: пропуски должны учитываться компилятором.

Многие правила стиля так или иначе связаны с расположением пропусков: новые строки до или после фигурных скобок, пропуски вокруг операторов, и т. д. Итак, я говорю, что эти языки должны учитывать пропуски.

И все же одна из вещей, которые нам вроде бы полагалось узнать из языков типа FORTRAN, что пропуски должны использоваться только для пометки границ между лексемами. Это считалось общим правилом, потому что в FORTRAN были столбцы: первые пять столбцов резервировались для номера команды или признака комментария, любой символ в шестом столбце был признаком продолжения предыдущей строки, столбцы 7–72 содержали программный код, а последние восемь резервировались для порядковых номеров, по которым было удобно собирать заново рассыпанные колоды. Да, колоды перфокарт — такие твердые, с прямоугольными дырками. Так что если символ попадал в неположенный столбец, команда могла превратиться в комментарий или еще что-нибудь, и это основательно раздражало. Кроме того, запись DO10I = 1,100 была эквивалентна DO 10 I = 1, 100 — DO было ключевым словом, за которым следовало число, а значит, пробел был не обязателен. Зато с записью DO10I = 1 все было еще интереснее, потому что она присваивала значение 1 переменной с именем DO10I.

Я пережил этот кошмар и испытал его на себе. Но это не доказывает, что пропуски должны игнорироваться. В действительности это доказало лишь то, что правила использования пропусков в FORTRAN были идиотскими. Свобода размещать пропуски где угодно на практике обернулась большими затратами. Программы уже не набиваются на перфокартах, а программы переформатирования стали такими же распространенными, как спам. Мы можем воспользоваться этим обстоятельством: вводите код, как хотите, но прежде чем компилировать — переформатируйте его. В конечном счете, необходимо только одно: чтобы редакторы и IDE дали возможность ввести программу и придали ей нужный вид. Фактически речь идет о переформатировании «на лету» — эта функция уже поддерживается многими редакторами. Никто не заставляет вас вводить 0, 1 или 17 пробелов между if и открывающей круглой скобкой; редактор (для C в стиле K&R) сам помещает туда ровно один пробел. Но даже эта задача становится еще проще, если будет только один стиль. Это одна из тех вещей, для которых могут пригодиться средства переформатирования или адаптации стилей. В сущности, свобода стиля форматирования обходится чрезвычайно дорого, причем эти затраты не окупаются. Подумайте, сможете ли вы честно заполнить следующее заявление:

> Мне, [*имя*], известен стиль программирования, влияние которого на производительность работы программиста и/или качество программ настолько велико, что мое право на выбор этого стиля вместо любых распространенных стилей оправдывает потери рабочего времени программистов и ресурсы, потраченные в масштабе отрасли на споры о стиле, выработку стилевых

> требований и переформатирование кода. Это стиль [*описание стиля*], и он обладает следующими преимуществами: [*описание преимуществ*].

И даже менее требовательное:

> Мне, [*имя*], известен стиль программирования, влияние которого на производительность работы программиста и/или качество программ > 5 % по сравнению с любым распространенным стилем. Это стиль [*описание стиля*], и он обладает следующими преимуществами: [*описание преимуществ*].

Думаю, у большинства из вас даже сама мысль о заполнении подобных заявлений вызовет лишь усмешку. И хотя в отдельно взятом проекте можно потратить 5 % на проблемы стиля программирования (в основном на начальной стадии), на этом хлопоты не прекратятся: стилевые войны вокруг того, что еще не было определено; предложения новых утилит, их написание или интеграция; внесение исправлений людьми, забывающими применить нужный стиль, с захламлением журнала изменений; обучение новых работников нужному стилю; вразумление непокорных; общая грызня, нытье и стоны.

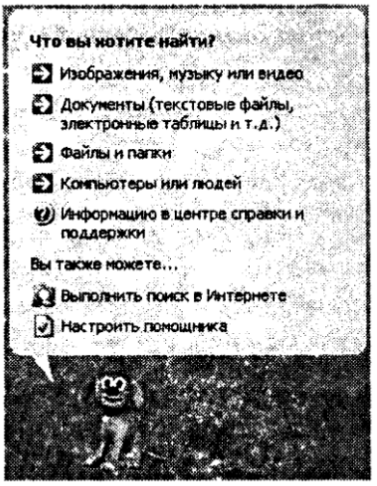
Так что 5 % даже отдаленно не покрывают затраты ресурсов и прочие неприятности, связанные с отсутствием обязательного стиля во всех программах в мире.

А может, вопрос лучше поставить наоборот: окупают ли преимущества от свободы стиля ту цену, которую мы за нее платим? Мне ответ кажется очевидным: не окупают *даже отдаленно*.

Леон Бамбрик

Премия за самый дурацкий пользовательский интерфейс: поиск в Windows

- Почему собака задает мне вопросы?
- Кто добавляет мескалин в напитки программистам Microsoft?
- А если бы такой подход использовался в Google?
- Остался ли бы Google «номером первым»?

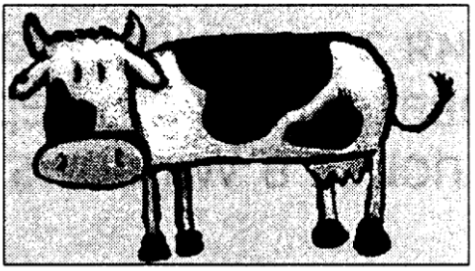


Google™

Захотели что-то найти?

- Помните, где в последний раз видели? **Да** или **Нет**
- Оно **больше** хлебницы? Или **меньше**?
- Это **животное**, **минерал** или **овощ**?

- Может, вам стоит **купить записную книжку**! Тогда не будете терять свои вещи.
- А **под кроватью** смотрели?



©2004 Google - Обработано 8 058 044 651 страниц

Если Microsoft не исправит этот нелепый подход к поиску, WinFS обещает стать настолько «навороченной», что пользоваться ей станет совершенно невозможно.

Майкл Бин

Подводные камни внешнего подряда

Почему некоторые производители программного обеспечения путают коробки с конфетами

Внешний подряд[1] (оффшорные и аутсорсинговые разработки) стали модной темой за последний год; наименее творческие компании в США стремятся переложить свои задачи по разработке приложений на умных, образованных программистов в Индии, Китае и Восточной Европе, а ставшие безработными американские программисты приводят искренние аргументы в пользу протекционизма.

Я читал целые книги, посвященные внешним разработкам. В сущности, никто из авторов так и не понял, что сущностью разработки программного обеспечения является проектирование, а не производство. Каждая написанная строка кода подразумевает решение относительно архитектуры программы. А для производителей программного обеспечения и всех остальных компаний, пользующихся конкурентными преимуществами от запатентованного программного обеспечения, передача проектирования на внешний подряд в конечном счете приведет к фатальным последствиям.

В 2001 году я писал:

«...Во время недавней мании доткомов некоторые невежественные авторы предположили, что компании будущего будут полностью виртуальными — стильные парочки будут попивать "шардонне" в гостиной, а вся работа будет выполняться внешними подрядчиками. Эти перевозбужденные "провидцы" упустили из вида одно обстоятельство: рынок платит за добавленную стоимость. Если двое яппи в гостиной покупают движок электронного магазина у компании A и продают товары, выпущенные компанией B, складируемые и поставляемые компанией C, клиентам, обслуживанием которых занимается компания D, — откровенно говоря, никакой добавленной стоимости они не создают. Каждый, кому хоть раз доводилось отдать на внешний подряд какую-нибудь критическую функцию, прекрасно знает, к какому аду это приводит. Без прямого контроля над обслуживанием клиентов технический сервис работает из рук вон плохо — почитайте, о чем пишут в блогах люди, безуспешно пытающиеся вызвать хоть кого-нибудь из телефонной компании для решения простейших задач. Если передать на внешний подряд исполнение заказов, и вы с вашим партнером расходитесь во взглядах на то, что считать "своевременной доставкой", клиенты тоже

---

[1] Также часто используются термины «удаленная работа» и «аутсорсинг». — *Примеч. перев.*

останутся недовольными — и с этим ничего нельзя будет поделать, потому что даже этого партнера на исполнение заказов пришлось разыскивать 3 месяца. К тому же вы не будете знать о возникшем недовольстве — ведь центр обслуживания клиентов был передан на внешний подряд именно для того, чтобы вам не приходилось слушать собственных клиентов. Купленный движок электронного магазина? Он ни при каких условиях не сравнится по гибкости с obidos, собственной разработкой Amazon. И никакой готовый веб-сервер не будет таким ослепительно быстрым, как вручную написанный и оптимизированный сервер Google».

Очерк Бина — самый ясный и наименее политически ангажированный труд на эту тему, который я когда-либо видел. Приятно видеть такое четкое объяснение, действительно проникающее в суть вопроса... особенно после огромного количества бреда, написанного на эту тему. — *Ред.*

Одежду и игрушки давно производят за границей. Так почему не перенести производство программ туда, где рабочая сила дешевле?

За последние годы многие технологические компании США перенесли разработку программного обеспечения в Индию. В 2004 году компания Hewlett-Packard стала крупнейшим индийским работодателем в области информационных технологий[1], с более чем 10 000 работников.

Энтузиазм по поводу внешнего подряда напоминает энтузиазм по поводу Интернет-компаний в 1990-х годах. Рави Чируволу (Ravi Chruvolu), партнер Charter Venture Capital, пишет: «Компании, специализирующиеся на венчурных инвестициях, решили, что из-за дешевизны технических специалистов в таких странах, как Индия, разработку программного обеспечения будет эффективнее передать на внешний подряд. Если компания Nike смогла перейти на внешнее производство кроссовок, мы можем сделать то же самое с программами»[2]. Следуя той же логике, компания Oracle решила удвоить количество программистов, работающих на нее в Индии[3], и довести его до 6000.

Многие авторы, пишущие о внешнем подряде, автоматически считают, что такие компании связываются этическими обязательствами по локальному сохранению рабочих мест. Хотя тенденция к внешнему подряду приводит к тому, что рабочие места уходят за пределы Соединенных Штатов, дело даже не в потере рабочих мест в США. Мы живем в условиях глобальной экономики. Жители Индии заслуживают работы в той же мере, что и жители Соединенных Штатов или любой другой страны; странно слышать, когда компании критикуют за внешний подряд только потому, что они нанимают людей за пределами своей страны.

Хотя внешний подряд не является неэтичным, разработчики программного обеспечения совершают стратегическую ошибку, когда они решают вынести разработку за пределы своей организации. Внешний подряд дает сбой тогда, когда фирмы-разработчики путают эксплуатационную эффективность со стратегией.

---

[1] См. http://news.zdnet.co.uk/business/employment/0,39020648,39118282,00.htm.

[2] См. http://www.charterventures.com/news/vcj_techtalk_2003mar.pdf.

[3] См. http://www.zdnet.com.au/jobs/news_trends/story/0,2000056653,20276211,00.htm.

Эксплуатационная эффективность направлена на уменьшение расходов или ускорение темпов работы. Стратегия направлена на создание долгосрочных конкурентных преимуществ; для компаний, занимающихся разработкой программ, к таким преимуществам относится возможность создания новаторских приложений. Внешний подряд в программировании работает тогда, когда разрабатываемая программа не является ключевой частью инновационного вектора для продуктов, продаваемых компанией. Например, перевод на внешний подряд программ обработки документации или дизайна веб-сайтов может быть оправдан, потому что он способствует повышению эксплуатационной эффективности.

Однако новаторские программы невозможно создать в условиях конвейера. Для них необходимы выдающиеся навыки проектирования и разработки. Передача разработки легионам зарубежных программистов приводит к потере преимуществ специализации. Передавая разработку на внешний подряд, компания утрачивает свои возможности по внедрению инноваций и свои конкурентные преимущества. Дело вовсе не в том, что индийские программисты уступают своим коллегам по компетентности или творческому мышлению. Внешний подряд мешает инновациям тогда, когда работники не могут общаться часто и спонтанно. При разности в девять часовых поясов такое общение становится невозможным[1]. Кроме того, может оказаться, что программисты, делающие открытия и придумывающие новые идеи для ваших программ, подолгу остаются недоступными, потому что они не работают на вас. Неважно, где расположена сама компания и где находятся программисты, работающие на внешнем подряде; если фирма-разработчик передает разработку базового программного обеспечения на внешний подряд, она не сможет обеспечивать инновации.

Если вы создаете передовую компанию, самых лучших и выдающихся программистов следует сохранить в штате. Разработчики программного обеспечения, расположенные полностью в Индии, способны к успешным инновациям в долгосрочной перспективе в такой же степени, как компании из США или любой другой страны. Плохой стратегией я считаю тенденцию передачи разработок на внешний подряд компаниями, базирующимся в США.

Так почему же некоторые производители программного обеспечения путают коробки с конфетами

Я живу неподалеку от Норт Бич в Сан-Франциско. Норт Бич славится своими итальянскими ресторанами, ночной жизнью и специализированными магазинчиками. Недавно я купил коробку шоколадных конфет в «XOX Trufles», одном из

[1] Разница в часовых поясах между Соединенными Штатами и Индией обычно означает отсутствие перекрытия между рабочими днями в каждой стране. В группах, координирующих свою работу по электронной почте, простой диалог из нескольких сообщений, занимающий 20 минут в одном часовом поясе, часто затягивается на несколько недель — ведь получатель сможет прочитать ваше сообщение только на следующее утро, когда вы спите. — *Примеч. перев.*

таких магазинов. Конфеты были просто потрясающие. Владелец магазина, Жан-Марк Горс, изготовляет их вручную, а его маленький магазин вошел в первую десятку по Соединенным Штатам.

Недавно Жан-Марк начал продавать конфеты в красивых коробках, синих с золотом. Мне понравилась его идея. Когда я спросил его насчет коробок, он ответил, что дизайн коробок разработала его жена, а он нашел компанию в Филиппинах, которая была готова выпускать коробки малыми партиями за хорошую цену.

Коробки Жан-Марка являются примером удачного внешнего подряда. Жан-Марк продает конфеты, а не коробки. Производство конфет находится исключительно в его компетенции. Жан-Марк находит внешнего подрядчика на производство коробок, повышая свою эксплуатационную эффективность без потери репутации производителя превосходных конфет. Но при этом Жан-Марк ни за что не доверит внешнему подрядчику производство своих шоколадных трюфелей, потому что это приведет к потере его преимуществ специализации. Тем не менее, в своем стремлении к сокращению затрат, многие производители программного обеспечения в США делают именно это — передают внешним подрядчикам свои ключевые технологии, обусловливающие их стратегические преимущества.

О различиях между проектированием и сборкой

Попытки поставить на поток разработку программного обеспечения предпринимались и раньше. В 80-х годах японские компании безуспешно пытались создавать «фабрики программирования» для массового выпуска программ. Оказалось, что собрать воедино множество программистов еще не значит создавать хорошие программы.

Но как я уже упоминал ранее, внешний подряд не всегда плох. А в некоторых отраслях он может оказаться абсолютно необходимым для сохранения конкурентоспособности. Например, производство одежды и игрушек на внешнем подряде

вполне оправдано. Основные затраты на производство одежды и игрушек обусловлены сборкой, а не проектированием. Эти продукты можно проектировать неподалеку от штаб-квартиры корпорации, а собирать за границей для снижения издержек.

Но написание программы в основном связано с проектированием. Практически все затраты на создание программного продукта обусловлены написанием программы, а не сборкой. Для программного продукта сборка в действительности сводится к записи окончательной версии продукта на диск и его упаковке вместе с руководством в коробку.

Майкл Портер (Michael Porter)[1] из Гарвардской школы бизнеса, эксперт по стратегиям и конкурентным преимуществам, хорошо сформулировал проблемы с конкуренцией, основанной исключительно на соображениях эксплуатационной эффективности[2]:

«Если вы фактически пытаетесь сделать то же самое, что и ваши конкуренты, вряд ли вас ждет большой успех. Со стороны компании было бы верхом самоуверенности полагать, что она может выпускать такой же продукт, что и ее конкуренты, и справляться с этим лучше в течение очень долгого срока. Это особенно справедливо в наши дни с невероятно быстрыми потоками информации и капиталов. В высшей степени рискованно рассчитывать на некомпетентность ваших конкурентов — а ведь именно это делает тот, кто делает ставку в конкуренции на эксплуатационную эффективность».

Стремление к внешнему подряду на разработку программного обеспечения наносит вред компаниям не из-за краткосрочных сокращений программистов, а из-за утраты компаниями способности к инновациям. В конечном счете им на смену придут конкуренты, использующие внутренний штат разработчиков и способные быстрее развиваться.

---

[1] См. http://dor.hbs.edu/fi_redirect.jhtml?facInfo=bio&facEmId=mporter.

[2] См. http://www.fastcompany.com/online/44/porter.html.

Рори Блайс

Excel как база данных

Еще немного, и корпоративные брандмауэры начнут вырезать графику из входящей электронной почты. — *Ред.*

Наверное, у любого разработчика в жизни был печальный момент (или несколько печальных моментов), когда ему передавали файл Excel, под завязку набитый «данными» каким-то спецом по маркетингу, и приказывали «что-нибудь сделать с этим».

Обычно в таких документах столбцы не выравниваются, а в документе задействована тысяча разных шрифтов. Создатель документа снова и снова злоупотребляет всеми возможностями Excel, чтобы только избежать использования приложения базы данных по прямому назначению — для хранения данных.

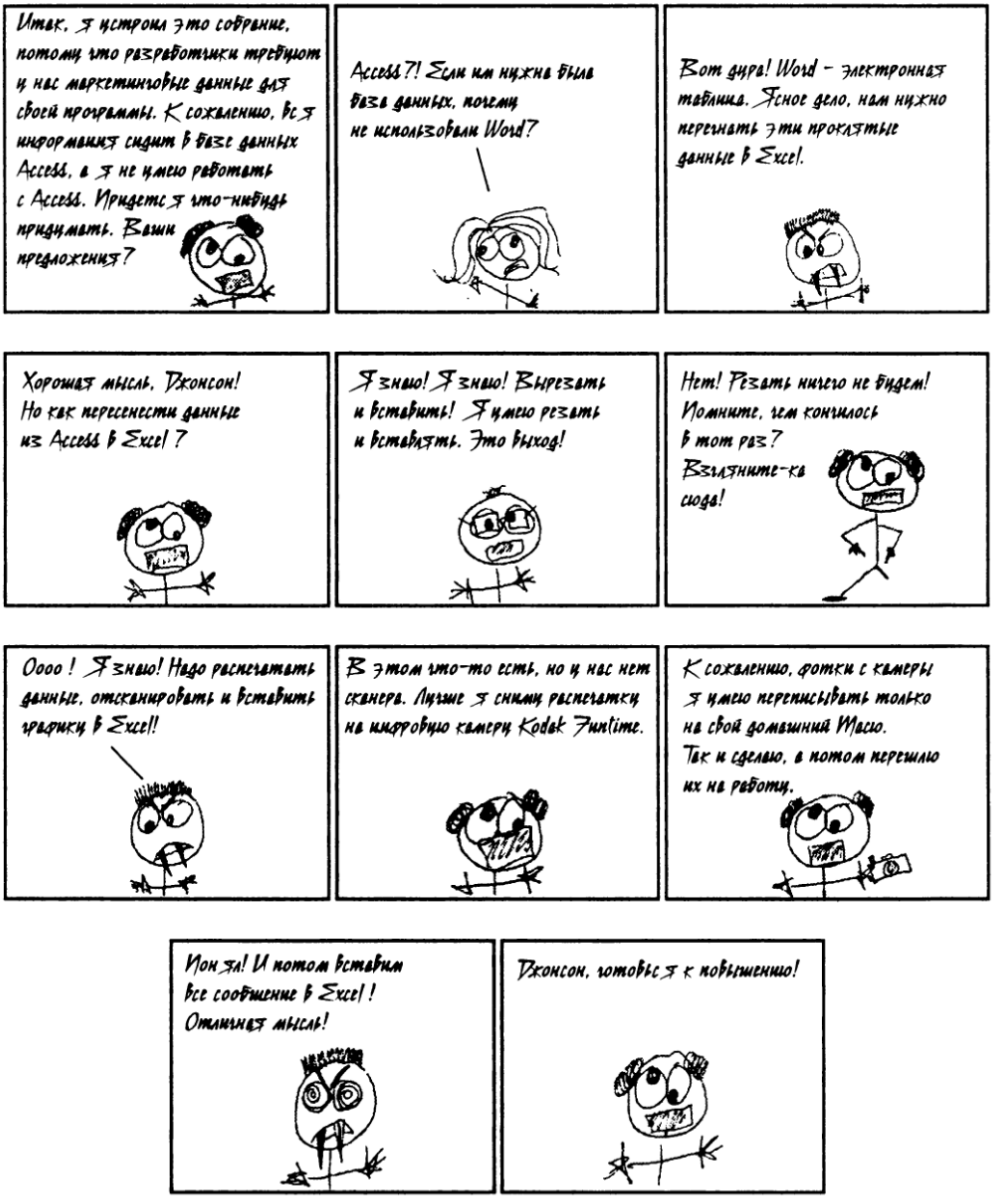
Конечно, вам предлагается самостоятельно разобраться в структуре документа, а когда вы жалуетесь на то, как трудно выковырять из Excel изуродованные данные и «что-нибудь сделать с этим», когда создатель документа не прикладывает ни малейших усилий по обеспечению логической целостности или хотя бы наведению элементарного порядка в данных, отдел маркетинга отделывается своей обычной [бип-бип].

Так родился мой высокохудожественный проект. Перед вашими глазами развернется настоящая драма — во всяком случае, в моем представлении подобные файлы создаются именно так.

- Обратите внимание: одного из персонажей я почему-то нарисовал с клыками и в конечном счете — с безумным взором. Не знаю, почему я это сделал. Просто показалось, что так будет правильно[1].

---

[1] Это потому, Рори, что ты совсем рехнулся. — *Примеч. ред.*



Знаю, что вы подумали — «Здесь ты, Рори, слегка загнул». Ну-ну.

Адам Босуорт

Доклад на ICSOC04

Адам Босуорт — самый выдающийся специалист по проектированию программных средств, о котором вы скорее всего, никогда не слышали, и один из ведущих мыслителей современности в области программных архитектур. Суть его выступления проста: умные теоретики разрабатывают изумительные, неимоверно жесткие и сложные концепции, которые попросту слишком сложны для обычных смертных... и поэтому никогда по-настоящему не прививаются. Но по-настоящему умные теоретики используют свой интеллект для упрощения своих творений, делают их доступными для широких масс, и именно такие архитектуры имеют значение.

Еще в 2002 году я написал[1]:

> Когда вам приносят спецификацию новой технологии, и вы в ней ничего не понимаете, не огорчайтесь. Все остальные тоже ничего не поймут, и, скорее всего, ничего путного из нее не выйдет. Вспомним SGML: никто не пользовался этим языком, пока Тим Бернерс-Ли радикально не упростил его, и народ внезапно все понял. По той же причине был упрощен протокол пересылки файлов, а на смену FTP пришел HTTP.
>
> Это повсеместное явление; даже в рамках одной технологии некоторые вещи достаточно понятны и используются на практике (скажем, IUknown в COM), а другие столь отвратительно усложнены (IMonikers), хотя могли бы быть простыми (чем плохи URL?), что в конечном счете увядают».

Проблема с излишне усложненными спецификациями заключается в том, что никто не хочет показаться глупым, и поэтому никто не предъявляет претензий к проектировщикам за то, что они спроектировали что-то слишком сложное. Пока C++ становился все более запутанным и невразумительным, никто не сказал: «Остановитесь, это слишком сложно для человеческого понимания», потому что люди не хотели выглядеть глупыми — но они молча «голосовали ногами», переходя на Visual Basic, PHP и Perl, понятные для простых смертных. Не огорчайтесь, если вы не можете разобраться в CORBA и не совсем понимаете, зачем нужны все эти WS-* — этого никто не понимает, и эти технологии вряд ли будут играть сколько-нибудь заметную роль за пределами очень узкой ниши. Технология VoIP

---

[1] Joel on Software, запись от 2 апреля 2002 г.; http://www.joelonsoftware.com/news/20020402.html.

долго оставалась невостребованной, потому что H.323 выходит за пределы понимания простых смертных. Но потом проектировщики Skype выкинули весь старый хлам и сделали штуковину, которая позволяет проводить телефонные звонки по Интернету. Неужели это было так сложно? — *Ред.*

Вчера я выступил с докладом на ICSOC04[1]. В сущности, я попытался напомнить группе очень умных людей, что их интеллект должен ориентироваться не на сложные, а на относительно простые пользовательские и программные модели. Поскольку передо мной выступал Дон Фергюсон из IBM, а на следующий день должен был выступить Тим Бернерс-Ли, мне показалось особенно разумным придерживаться простых и понятных идей. Приведу текст своего выступления.

До и после меня выступают куда более умные и заслуженные докладчики. Дон Фергюсон из IBM создает концепции настолько изощренные и хитроумные, что перед ними меркнут труды архитекторов, создавших знаменитые арки Альгамбры. Тим Бернерс-Ли создал Всемирную Паутину в том виде, в котором мы ее знаем, а сегодня он из своего орлиного гнезда в MIT проповедует своего рода религию семантической Паутины, совершенно недоступную для моего понимания. Эти джентльмены чрезвычайно умны. Было бы глупо пытаться выглядеть умным, выступая между ними. Соответственно, я пойду в обратном направлении и поговорю о достоинствах подхода KISS[2] и его влиянии на технологии Интернета.

Бесспорно, существуют извечные разногласия между частью человечества, считающей наше разнообразие, несовершенство и ошибки особой частью богатого спектра человеческой души, и другой частью, стремящейся к совершенству, полному контролю, построению сложных систем и правил, тщательное соблюдение которых гарантирует полную упорядоченность процесса. Мой доклад посвящен этому конфликту в контексте Интернет-технологий. Кроме того, прошу считать его полемическим выступлением в поддержку KISS. Как таковой, мой доклад необъективен, отражает мою личную точку зрения, а местами даже использует некорректные приемы. В самом деле, порой он балансирует на грани ламентации.

По иронии судьбы попытки создания систем, рассчитанных на совершенство человеческой натуры, в конечном счете, приводят к созданию систем, наиболее бездушных и жестких — систем, которые загнивают изнутри, пока словно величественные, скрипящие и трухлявые дубы, не обрушиваются наземь, распространяя кислый запах тления. Мы видели, как это произошло в 1991 году с невероятным падением СССР. И наоборот, системы, учитывающие сложность, хрупкость, великолепие человеческой натуры, в основу которых заложена гибкость, многократные проверки и допуски, часто выживают вопреки всем прогнозам. Так же дело обстоит и с программным обеспечением. Программы гибкие, простые, терпимые, снисходительные к человеческим слабостям, оказываются самыми устойчивыми, приспособленными к выживанию и развитию — а программы требовательные, абстрактные, насыщенные возможностями, но излишне систематизированные, ждет медленный и неотвратимый развал. Возьмем электронную таблицу. Это изменчивая, неформальная, пластичная, гибкая среда, которая по иронии судьбы

---

[1] См. http://icsoc.dit.unitn.it.

[2] Сокращение от «Keep it Simple, Stupid», то есть «Простота — залог успеха». — *Примеч. перев.*

превращается в настоящий кошмар для всех бухгалтеров и аудиторов, потому что в действительно сложной и функциональной электронной таблице разобраться практически невозможно. Компания Lotus Corporation (ныне IBM), призвав на помощь магистров из Гарварда и докторов в области информационных технологий из MIT, создала Improv — программу, которая обещала «решить все проблемы». Это была настоящая мечта аудитора. Программа обеспечивала неслыханные высоты абстракции и формализма представления строк и столбцов... короче говоря, была воистину всесторонней. Однако ее ждал полный провал, и вовсе не потому, что программа не сдержала своих обещаний — как раз наоборот[1].

Возьмем поиск. Я помню первые неуклюжие демонстрации Microsoft, когда Билл Гейтс впервые завел речь об «информации под рукой»[2] — сложные экраны для ввода критериев поиска, поддержка булевской логики. В одном из моих собственных продуктов, Access, была реализована вроде бы более простая технология QBE (Query By Example)[3]. Сегодня полмиллиарда людей ежедневно пользуются поисковыми системами, и что же они применяют? Не QBE. Не булевскую логику. Они выбирают потрясающе элементарную и неоднозначную технологию, а именно произвольный текстовый поиск. Реализация сложна, но пользовательская модель проста и тривиальна.

Возьмем пользовательский интерфейс. При первом появлении HTML эта технология была фантастически неформальной, нечеткой и либеральной. Помню, еще в 1995 году руководитель проекта Microsoft Office презрительно говорил, что HTML никогда не будет иметь успеха из-за своей примитивности, а Word победит, потому что документы Word так богаты возможностями, а их структура полностью формализована. Конечно, HTML в наши дни стал основным строительным материалом для огромных хранилищ человеческой информации. Более того, это один из непредвиденных парадоксов в истории компьютерных технологий: изначально предполагалось, что HTML будет использоваться как язык пластичной разметки, не подверженный двумерным ограничениям — парадокс, потому что орды фанатиков CSS с тех пор пытаются упаковать его в смирительную рубашку и проклинают таблицы, а целые поколения программных инструментов создают на его базе

---

[1] Хорошо помню Improv; программа вышла как раз тогда, когда я работал над Excel, и обещала стать «будущим электронных таблиц». Вместо отображения удобной сетки с ячейками Improv требовала определять n-мерные гиперкубы для представления данных. Вместо введения формулы в любой ячейки разрешалось только определять новые строки и столбцы, значения которых вычислялись на основании существующих строк и столбцов. От гибкости традиционных электронных таблиц не осталось и следа. Подразумевалось, что электронные таблицы используются для обсчета математических моделей, с которыми имеют дело аспиранты. Небольшое маркетинговое исследование показало, что большинство пользователей применяет электронные таблицы для простого ведения списков, а реальный мир укладывается в n-мерные гиперкубы совсем не так хорошо, как казалось в Уортоне. — *Примеч. ред.*

[2] IAYF («Information At Your Fingertips») — «предвидение», на реализацию которого Билл Гейтс направил Microsoft в начале 1990-х, до пришествия Интернета. — *Примеч. перев.*

[3] Интерфейс для поиска в базе данных: пользователь заполняет новую запись, у которой одни поля заполняются, а другие остаются пустыми. После нажатия кнопки ядро базы данных возвращает набор всех записей, удовлетворяющих поставленному критерию. Скажем, если ввести в поле возраста значение «<18», а в поле штата — «NY», вы получите полный список всех жителей штата Нью-Йорк моложе 18 лет. — *Примеч. ред.*

многослойные двумерные разметки с точностью до пикселя. И все же спросите любого одаренного веб-дизайнера — такого, как Джон Уделл (Jon Udell), и он вам ответит, что часто использует HTML тем неформальным, нечетким, интуитивным способом, для которого он и проектировался. Они просто «вливают» в HTML содержимое. В 1996 году я присутствовал на одном из первых семинаров XML. Участники метали громы и молнии в HTML за то, что он «загрязняет» содержимое разметкой. Среди первых поклонников XML было много разочарованных любителей SGML, возжелавших лучшего и совершенного мира, в котором данные безукоризненно отделялись бы от представления. Короче, они протестовали против одной из выдающихся историй успеха в компьютерных технологиях — причем той, которая стала возможной благодаря своим недостаткам, а не вопреки им. Я сильно сомневаюсь, что HTML, который бы изначально распространялся с четкой многоуровневой иерархией (XML, правила разметки — XSLT, и форматирование — CSS), пользовался бы столь оглушительным успехом.

В 1996 году я поддерживал XML, но как оказалось, я поддерживал его по прямо противоположным соображениям. Мне хотелось иметь гибкий, неформальный, «человеческий» способ обмена данными между программами, а по сравнению с RPC, DCOM и IIOP тех дней XML был невероятно гибкой, пластичной и естественной средой. И до сих пор остается. И именно по этой причине, а не вопреки ей, XML быстро завоевывает место общепринятого механизма обмена данными между программами. А старые системы медленно, но верно обрушиваются и уходят в никуда.

Возьмем само программирование. Каждый день в мире программирования идет необъявленная война. Это война между простыми людьми и теоретиками. Это война между теми, кто желает простых, неформальных, гибких, человеческих средств программирования, и теми, кто ищет четких, однозначных, единственно правильных решений. Это война между PHP и C++/Java. Раньше это была война между C и dBASE. Учащиеся Колумбийского университета; те, кто прошел через жесткий отбор службы найма Google; другие программисты этого уровня — все они любят точные инструменты, абстракцию, стройные ряды упорядоченных концепций и формальную логику. Но большинству программистов ближе позиция моего сына. В их представлении программный код сродни молотку, который используется для выполнения работы. PHP для них является идеальным языком. Он прост. Он высокопроизводителен. Он гибок. Основой этого языка являются ассоциативные массивы, поэтому он, как и XML, является гибким и самодокументирующим. Это позволяет легко написать код, который динамически адаптируется к переданной информации и легко выдает нужный XML или HTML. Для таких людей важно содержание и сообщество, а не тонкости технологии. Как найти нужные поставки RSS? Как организовать взаимодействие сообщества, назначить модераторов и динамически принимать решения относительно того, какие сообщения можно пропускать, а какие должны подвергаться предварительному просмотру? Как отфильтровать информацию по репутации? Их интересуют именно эти вопросы, а не сам язык.

Аналогичным образом, я наблюдаю две диаметрально противоположные тенденции в современных моделях обмена информацией между программами. С одной стороны мы видим RSS 2.0 и Atom. Документы, основанные на этих форматах, плодятся как кролики. На самом деле никого особенно не интересует, какой именно

формат используется в конкретном случае, потому что они во многом взаимозаменяемы. Оба формата практически представляют собой списки ссылок на содержание с прикрепленными метаданными. Оба формата делают возможной реализацию моделей сохранения репутации, фильтрации, выносной аннотации и т. д. Известна бесплодная попытка провести для этого сообщества жесткую абстрактную формализацию под эгидой RDF и RSS 1.0. Попытка завершилась неудачей. Это произошло потому, что она была слишком абстрактной, слишком формальной и одновременно слишком сложной для «ударных частей», которые просто пытаются решить конкретную задачу. Вместо этого победили RSS 2.0 и Atom; сейчас они широко используются для проведения ток-шоу и формирования списков воспроизведения (podcasting), фотоальбомов (Flickr), расписаний событий, списков интересных ресурсов, новостей и т. д. Воспроизведение доступно каждому. Технология становится простым, неформальным *lingua franca*, по которому распространяется информация в Web. Во время распространения она фильтруется, обобщается, расширяется и даже преобразуется в новые формы, как вода, текущая из ручьев по рекам к огромным заливам. Эту информацию можно получить напрямую по HTTP, указав URL. В большинстве языков для ее выборки достаточно ввести одну строку кода. Это тот мир, к которому благополучно адаптируются Google и Yahoo, при всей своей пластичности, гибкости, непредсказуемости, и в то же время простоте и ориентации на конечного потребителя.

С другой стороны, мы имеем мир SOAP, WSDL, XML SCHEMA, WS_ROUTING, WS_POLICY, WS_SECURITY, WS_EVENTING, WS_ADDRESSING, WS_RELIABLEMESSAGING и других попыток формализации моделей расширенного информационного обмена (rich conversation). Каждая из этих спецификаций толще и гораздо сложнее исходной спецификации XML. Этот мир в высшей степени удобен для IT-отделов крупных корпораций. Он кажется стопроцентно контролируемым и подотчетным. Если мир RSS можно сравнить с ручьями, реками и заливами, воды которых несут смытые частицы донных отложений, этот мир является миром Шлюзов, Водоводов, Дамб и Фильтров Чистой Воды. Это мир для экспертов — сокровенный, сложный и эзотерический. Код обработки этих сообщений проходит настолько раннее связывание, что он предварительно компилируется на основании WSDL, и как многие узнали на собственном опыте, если он не работает — ни одна человеческая душа не выяснит, почему. Трудно передать словами, насколько велики различия между HTTP с его небольшим количеством простых команд и этим миром, бесчисленные уровни которого объединяются в неслыханно сложный механизм. Короче говоря, это мир, который могут полюбить только IBM и Microsoft. И они его полюбили.

С одной стороны мы имеем блоги, фотоальбомы, расписания событий, рейтинги и поставки новостей. С другой — CRM, ERP, BPO и кучу других трехбуквенных сокращений, ориентированных на корпоративную среду.

Как я уже говорил, много лет назад некто заявлял, что HTML никогда не будет иметь успеха из-за своей примитивности. Конечно, HTML ждал успех именно по этой причине — потому что он был примитивен. Сегодня я слышу, как те же люди в тех же компаниях говорят, что технология XML по HTTP никогда не будет иметь успеха из-за своей примитивности. Успех возможен только с SOAP и SCHEMA. Но настоящее волшебство XML заключается в том, что этот язык является само-

документирующим. Разработчики RDF этого так и не поняли, потому что они искали то, что найти нельзя — абсолютную истину. Сказать, что XML не может иметь успеха из-за неизвестной семантики — все равно, что сказать, что реляционные базы данных не могут иметь успеха из-за неизвестной семантики, или текстовый поиск не может иметь успеха по той же причине. Однако в этом утверждении имеется зерно истины. И раньше, и сейчас трудно высказать об XML какую-то непреложную истину. Вот почему InfoPath[1] пришлось идти на такие выкрутасы, чтобы сделать возможным простое редактирование. Напротив, модель RSS проста благодаря почти произвольному набору известных свойств для элемента списка: имя, описание, ссылка, тип MIME и размер для вложения. Как и в HTML, определяется минимальная полезная информация. Как и в HTML, ее можно расширить в случае необходимости, но большинство людей подходит к этой возможности весьма осмотрительно. Поэтому блог-ридеры и агрегаторы могут без особых проблем отобразить содержимое, понимая, что ценность заключена в информации. Да, между блог-ридерами и InfoPath есть еще одно различие: блог-ридеры бесплатны. Их авторы понимают, что ценность кроется в содержимом, а не в механизме для его просмотра[2].

В RSS воплощен очень простой принцип, который Тим Бернерс-Ли всегда считал одним из самых важных и центральных догматов своей революции: а именно, что к любому элементу содержания можно обратиться по URL. На языке RSS это называется «перманентными ссылками» (permalinks)[3]. Эта идея имеет глубокое значение. Дэйс Сифри (Dave Sifry) из Technorati недавно обратил мое внимание на то, что одним из самых замечательных аспектов RSS и веб-журналов (блогов) является их способ решения одной из самых трагичных проблем Web, а именно нецивилизованности общения. Web в своем сегодняшнем виде во многих отношениях напоминает классический пример «трагедии общественных земель»[4]. Поскольку рассылка электронной почты практически ничего не стоит, появился спам. Поскольку отправка сообщений осуществляется практически бесплатно и анонимно, появились группы, в которых небольшая группа людей может заглушить обсуждение назойливой и бессмысленной болтовней. Но одно из достоинств возможности адресации каждого элемента заключается в том, что комментарии по поводу элементов могут распространяться в Web. Web становится чем-то вроде огромной комнаты, в которой люди комментируют мысли других людей, используя для этой цели сообщения в своих веб-журналах. Поступая таким образом, они подвергают риску свою репутацию. Вряд ли это можно назвать дешевыми и анонимными сообщениями — скорее это недвижимость в пространстве, связанном с вашей личной точкой зрения и репутацией. По этой причине комментарии обычно становятся

---

[1] Продукт Microsoft для заполнения форм; в сущности, приукрашенный редактор XML. — *Примеч. перев.*

[2] Во всяком случае, некоторые. InfoPath стоит так дорого, что он не вышел за пределы корпораций, которые уже выплачивают крупные ежегодные отчисления Microsoft за полную версию Office. — *Примеч. ред.*

[3] Когда блоггер хочет создать ссылку на нечто, написанное сегодня другим блоггером, то вместо ссылки на его домашнюю страницу, которая завтра изменится, он использует «перманентную ссылку», по которой всегда отображается одно и то же содержание. — *Примеч. ред.*

[4] См. http://en.wikipedia.org/wiki/Tragedy_of_the_commons — *Примеч. перев.*

более взвешенными, продуманными и осмотрительными. А если этого не происходит, участники либо решают, что их это устраивает, либо отказываются от дальнейшего участия. Происходящее можно сравнить с «дуэлью передовиц» в паре газет.

Напротив, жесткие абстрактные уровни архитектуры веб-служб представляют лишь анонимные, бесконечные сообщения, проходящие под разрядом одного URL. Если сообщения не регистрируются в журнале, никакой ответственности нет и в помине. Так как спецификация XML Schema, определяющая их грамматику, представляет собой гибрид верблюда со слоном и жирафом, эти сообщения могут понравиться разве что специалисту по африканской фауне. Впрочем, имейте в виду, что даже эти сообщения гораздо лучше предшествовавших им сообщений MOM[1]. Поскольку сообщения являются самодокументирующими, вы можете установить динамические фильтры для изменения их маршрута или структуры с использованием XPATH, XSLT, XML Query и даже других языков; все эти решения позволяют легко проверить, является ли сообщение существенным, и если является — выявить его интересные части. Все это хорошо и достойно двадцать первого века. Но точки отправления и приема, окутанные немыслимыми сложностями JAX RPC и .NET, все еще застыли в жесткости раннего связывания[2] двадцатого века.

Мне хотелось бы сказать, что мы находимся на распутье, но истина никогда не бывает такой простой. А правда в том, что люди используют те средства, которые им подходят. Просто для некоторых программистов подходящим инструментом является PHP, а для других — Java, поэтому также будет справедливо сказать, что для некоторые программистов подходящим инструментом является RSS, а для других — WS-*. Здесь нет однозначного «победителя». Уверен я лишь в ценности информации и возможности ее беспроблемного объединения, оценки, фильтрации и расширения.

Что это означает лично для вас? Вспомните радио. Когда оно было техническим новшеством, настоящей ценностью был сам радиоприемник. Содержания было относительно немного, но многие люди хотели иметь дома собственный приемник. Однако в какой-то момент качество самих аппаратов и передачи данных стало достаточно высоким, и ценность радио неизбежно сместилась в область содержания. Именно поэтому идут такие ожесточенные баталии вокруг DRM[3], феномен podcasting произвел такую революцию, а Говарду Стерну так много платят за выступления на спутниковом радио. Ценность кроется в содержании. Мы достигли примерно той же точки в компьютерных технологиях. Их ценность заключена уже не в компьютерах и не в программах, которые на них работают, — она в содержании, в умении программ находить и фильтровать это содержание, а также в возможностях программ по организации содействия людей и обмена информацией о содержании (а также друг о друге). Никого не волнует, если в Excel появится новая команда меню — если только она не упростит поиск информации в Web,

---

[1] Messaging-Oriented Middleware. Не знаю, что это такое, но звучит кошмарно. — *Примеч. ред.*

[2] Видимо, говоря о «раннем связывании», Адам имеет в виду, что написанный код предполагает, что формат всех сообщений известен заранее. Код компилируется для работы только с сообщениями именно этого формата, а если что-то вдруг изменится — происходит исключение, и программа перестает работать. — *Примеч. ред.*

[3] Digital Rights Management, то есть «Управление цифровыми правами». — *Примеч. перев.*

а возможно, и ее обновление и обмен информацией с другими пользователями по поводу вашей находки.

Как насчет мобильных телефонов? Кому-нибудь захочется иметь на мобильном телефоне электронную таблицу или PowerPoint? Или все же будет интереснее узнать, где находится ближайший банкомат или ресторан индийской кухни, где продается литература по компьютерным технологиям нужной тематики, и что пишут в рецензиях на эти книги? Действительно ли интересно иметь адресную книгу, синхронизированную с PC, или интереснее видеть присутствие людей, участвующих в ваших занятиях, в вашем проекте, в ваших планах на вечеринку, и иметь возможность координировать и планировать событие вместе с ними? А если вы выберете второй вариант, разве ценность в действительности происходит не от знания тех, с кем вы работаете, общаетесь и учитесь; того, что они думают об интересующих вас вещах, будь то кино, классы, рестораны или новости; а не от программ на самом устройстве? Как только вы вливаетесь в Web, 2 миллиарда людей смогут увидеть вашу информацию и взаимодействовать с ней.

Сейчас идет много разговоров о Web 2.0. Похоже, многие полагают, что во «второй» Паутине центральное место займут развитые интеллектуальные клиенты, обменивающиеся информацией через Web и имеющие дело с мультимедийным содержанием (фото, звук, видео). Несомненно, так оно и будет. Будь то Skype[1], наш продукт Hello[2] или iTunes[3], люди все чаще подключаются к Web для обмена мультимедийным содержанием. Однако я утверждаю, что это изменение не является существенным. Оно эффектно, занятно, полезно, но в конце концов, ничего принципиально нового в нем нет.

Принципиально новым является фактор информационной перегрузки. Электронная почта уже давно стала настоящим проклятием. Блог-ридеры только усугубляют проблему. Я не могу даже отдаленно представить нечто аналогичное в области видео- или аудио из-за гораздо больших трудностей с фильтрацией. Принципиально новым будет то, что люди будут собираться для того, чтобы оценить, обсудить, проанализировать и реализовать 100 000 предложенных Zagat[4] моделей доверия к информации, товарам и услугам. Мы уже видим нечто похожее на eBay. Мы видим возрастающую роль количества сделок и рейтингов людей, продающих книги на Amazon. Как я писал в своем блоге:

> «...Моя мать ни разу не жаловалась, что ей нужен более качественный клиент для Amazon. Вместо этого ее интересуют более удобные средства общения, более качественные списки книг, более простые средства их просмотра, более надежная информация от рецензентов, мнение библиотекарей (потому что она сама работает в библиотеке), и т. д.»

Вот это действительно будет новым. А в сущности, уже есть. Хотите заглянуть в будущее? Не смотрите на Longhorn. Взгляните на Slashdot: 500 000 «нердов» ежедневно собираются вместе только для того, чтобы справиться с информаци-

---

[1] Бесплатный сервис Интернет-телефонии. — *Примеч. перев.*

[2] Приложение Google для обмена графикой. — *Примеч. перев.*

[3] Онлайновый музыкальный магазин Apple. — *Примеч. перев.*

[4] Путеводитель по ресторанам, в котором отзывы предоставляются читателями. — *Примеч. перев.*

онной перегрузкой. Взгляните на Bloglines[1]. Каким будет решение проблемы? Будет ли это Attention.XML[2], как надеются Стив Гилмор (Steve Gillmor) и Дэйв Сифри (Dave Sifry)? Или что-то менее формальное и более органичное? Неважно. Актуальность репутации и оценок станет ответом на проблему «трагедии общинных земель», и выход будет найден.

Именно здесь будут происходить основные события. Будет совершенно неважно, знаете ли вы Avalon[3] или Swing[4]. Важными будут самообучающиеся машины, методика логических построений и информационная проходка (data mining). Впервые с момента появления компьютеров искусственный интеллект находит массовое применение. Меня это явление в высшей степени радует. Оно означает, что в конечном итоге ценность кроется в нашей человеческой натуре, в нашем разнообразии и сложности, в нашем умении сотрудничать. Оно означает, что на первый план выходят человеческие, гибкие, органичные аспекты Web, в отличие от аспектов сухих, аналитических, таксономических.

Я чувствую глубокий оптимизм и надежду на то, что рост Web медленно повысит уровень и содержательность общения. Подобно порнографии, которая, как ни печально, стала одной из ведущих областей применения технологии, грубое и оскорбительное общение было взято за образец для подражания ее ранними последователями. Отрадно видеть, что начали появляться первые признаки самоорганизации, доверия и осмотрительности в общении.

Кто-то скажет, что все это утопия. Если называть «утопистом» человека, который не боится мечтать — так оно и есть. Такой же утопией было исходное представление Тима о всеобщем доступе к информации. Такой же утопией является цель Google. Лоуренс Аравийский писал в «Семи столпах мудрости»: «Все люди мечтают; но неодинаково. Те, кто мечтает по ночам в тайниках своей души, утром просыпаются и обнаруживают, что все это тщетно. Но те кто, мечтают днем, опасные люди, потому что они могут сделать свои мечты явью».

Я призываю всех вас мечтать днем, с открытыми глазами. Я призываю вас всех мечтать об Интернете, который бы давал людям возможность вместе работать, общаться, помогать друг другу и делать совместные открытия. Я призываю вас помнить, что, в конце концов, все мы люди, и если вы создаете нечто новое — создавайте его простым, гибким, неформальным, и в конечном счете наделенным всеми качествами, столь ненавистными для наших внутренних платоников.

---

[1] Онлайновый объединитель блогов, в настоящее время принадлежащий InterActivecorp. — *Примеч. перев.*

[2] Стандарт XML для отслеживания источников информации, заслуживающих внимания. — *Примеч. перев.*

[3] Библиотека графического программирования для запланированной операционной системы Microsoft «Longhorn». — *Примеч. перев.*

[4] Библиотека графического программирования для Java. — *Примеч. перев.*

Дана Бойд

Аутизм в социальном программном обеспечении

Наверное, год 2004 войдет в историю как год, в котором «ботаники» из Кремниевой Долины с их социальными дисфункциями начали привлекать капитал к кодированию своего синдрома Аспергера в социальных интерфейсах. Эти интерфейсы создаются ими на базе Orkut и LinkedIn и демонстрируют, что их разработчики совершенно не понимают взаимодействия между людьми, не говоря уже о взаимодействиях при посредничестве компьютера. Недавно я узнал из блога Даны, что AOL позволяет иметь только 200 друзей. Прежде всего, что это за число такое — 200? Это даже не степень двойки! Что происходит! Я прямо-таки слышу Дастина Хоффмана в «Человеке Дождя»: «Нельзя иметь больше 200 друзей. Нужно удалить друга. Это плохо». — *Ред.*

Мы, технические специалисты, часто ставим во главу угла применение технологий вместо того, чтобы строить технологию на базе привычек и потребностей пользователя. В этом докладе я сделаю шаг назад и предложу иной взгляд на то, что мы, техники и бизнесмены, сделали, и какие ценности мы сформировали в пользователях.

Моя цель — заставить нас изменить свой подход, чтобы он был действительно направлен на удовлетворение людских потребностей.

Среда общения, научная фантастика и психические расстройства

Хотя «социальное программное обеспечение» принадлежит к числу недавних явлений технического сообщества, понятие «среды общения» существует с первых дней Интернета. Электронная почта, BBS, Usenet, чат, MUD и MOO прочно захватили воображение изобретателей новых технологий в 1980-е и 1990-е годы. Параллельно с разработкой этих технологий академики и провидцы вещали о том, какие утопические мечты могут быть воплощены благодаря этим новшествам. В их речах отражались концепции, предлагавшиеся научной фантастикой, но нередко без богатого воображения авторов-фантастов. Идеалисты представляли себе мир, в котором

телесная оболочка становится абсолютно несущественной, потому что в виртуальном мире никто не будет знать, что вы представляете собой на самом деле.

Хотя многие авторы научной фантастики пытаются передать нюансы человеческого поведения, их основное внимание сосредоточено на сюжете, и социальные аспекты, связанные с технологиями, часто излагаются лишь в той мере, в которой они затрагивают сюжет. Построение глобальных предположений на базе ограниченных сценариев, предлагаемых научной фантастикой, — дело сомнительное, поскольку при этом утрачивается замечательное разнообразие человеческого поведения.

Научная фантастика не пытается понять человеческую психологию в целом; авторы всего лишь пытаются использовать некоторые аспекты человеческого поведения для своего повествования.

Расширение этих концептуальных моделей до мира в целом не может устоять в столкновении с реальностью, потому что наши жизни не укладываются в четкую линию повествования. С точки зрения человеческой психологии научно-фантастические модели часто наивны и упрощены, они являются инструментами для развития сюжета. За пределами фантастики человеческая психология была темой культурного дискурса в течение двух последних десятилетий, а темы человеческих дисфункций и психических расстройств проникли в массовое сознание через научно-популярные статьи и фильмы. Вспомните, что Джордж Буш старший объявил 1990-е годы «Десятилетием мозга».

Хотя в популярной литературе нашли отражение все типы психических расстройств, тема расщепления личности особенно захватила воображение публики в 1980-е и 90-е годы. Расщепление личности стало восприниматься как каноническое психическое заболевание, и некоторые фильмы попытались воспроизвести его сущность.

Обсуждения человеческой психологии, психических расстройств и расщепления личности также появлялись в Интернете. Сэнди Стоун (Sandy Stone) и Шерри Теркл (Sherry Turkle), два известных исследователя сред общения, анализировали возможности, открываемые цифровыми взаимодействиями в контексте расщепления личности. Они увидели возможность ведения «параллельных жизней» и создания «альтернативных личностей» как перспективу освобождения субъекта от ограничений физического тела в повседневной жизни.

Технологии сетевого общения не только поддерживали, но и поощряли применение псевдонимов; даже сегодня это средство рассматривается как защита от вторжения в личную жизнь. Людей подталкивают к фрагментации личности на псевдонимы с целью должной контекстуализации их сетевого общения. Фактически поощряется формирование «альтернативных "я"».

И знаете, что оказалось? В реальности люди не настолько фрагментированы. Хотя они могут вести многогранную жизнь, их контроль над тем, какую информацию о себе следует предоставлять в каждом конкретном случае, слишком тонок для простого деления на несколько личностей. К сожалению, наши ранние представления о расщеплении личности проникли не только в разговоры, но и в реальные технологии сред общения. Когда бы я ни выражала беспокойство о конфиденциальности или уязвимости личности, мне часто говорили, что людям просто следует создать несколько виртуальных личностей.

Только подумайте, как это глупо. Ради чего заставлять людей имитировать психическое расстройство в цифровом мире? Мы делаем это потому, что построенная нами технология не учитывает нюансов многогранности личности, к которой уже привыкли люди. Нас, технарей, на курсах системотехники учили отделять политику от механизмов. Мы предполагаем, что люди сами разберутся с политикой, не понимая, что мы ограничили их возможности своими технологиями.

Как выяснилось, Интернет так и не превратился в фантастический мир, в котором социальная личность перестала играть какую-либо роль. В проекте «Игра Тьюринга»[1] Эми Брукман (Amy Bruckman) показала, что люди проявляют свою повседневную личность через виртуальные персоны даже в том случае, когда они стараются действовать иначе. В наши дни существуют технологические разногласия между интегрированными системами подтверждения личности[2] (такими, как Passport) и продолжающейся тенденцией к построению систем, в которых пользователи создают новую личность для каждой системы. Дебаты на эту тему имеют псевдорелигиозную окраску, но все усилия, о которых мне известно, попрежнему концентрируются на технологии, а не на людях и практике. Из-за этой смены фокусировки такие вещи, как списки контроля доступа (ACL) и открытые протоколы FOAF (Friend of a Friend) обречены на неудачу. Они не имеют аналогов в повседневной жизни.

Аутизм и синдром дефицита внимания

Ранние среды общения были скованы представлениями научной фантастики и метафорами популярной психологии, но и современные среды общения не избавлены от этих стереотипов. Социальные расстройства, хотя и несколько иные, продолжают влиять на наши представления о человеческой психологии.

Возьмем многочисленные статьи об аутизме и синдроме Аспергера в 2003–2004 годах. Для тех, кто не слышал о синдроме Аспергера, напомню: это умеренная форма аутизма, для которой характерно нормальное развитие интеллекта с ограниченными социальными и коммуникативными навыками. Пациенты с синдромом Аспергера часто стремятся систематизировать социальную деятельность для придания ей структуры, необходимой для ее процедурного выполнения в повседневной жизни. Важно заметить, что синдром Аспергера часто объединяется с другим «психическим расстройством», любимым средствами массовой информации: синдромом дефицита внимания (СДВ). Для синдрома дефицита внимания часто характерна невозможность сосредоточиться на выполняемой работе, склонность к чрезмерному сосредоточению с последующей полной потерей концентрации. Как и в случае с расщеплением личности, из-за средств массовой информации аутизм и СДВ стали выглядеть явлением обыденным и повсеместным.

---

[1] См. http://www.cc.gatech.edu/elc/turing/.

[2] Гипотетическая система, которая никогда не была успешно реализована на практике; представляет собой единую базу данных с полной информацией обо всех учетных записях, позволяющую использовать один набор регистрационных данных для обращения к любому веб-сайту и учетной записи. — *Примеч. ред.*

Разработчики технологий тоже приняли эти концепции и способствовали их развитию, отметив их ценность для стиля жизни «настоящего технаря». Многие из вас сейчас смотрят на экраны своих ноутбуков, работая в многозадачном режиме[1]. Хотя они запомнят только часть моего доклада, вероятно, они скажут, что запомнили самое важное, или что они практикуются в модном «перманентном частичном внимании». Возможно, среди них действительно найдутся такие знатоки, и все же мне кажется, что большинство слушателей не уделяет внимания ни мне, ни компьютеру. Но они хотят быть мастерами перманентного частичного внимания, потому что им сказали, что это характерно для всех крутых парней. Конечно, аутизм куда менее эффектен, чем СДВ, но некоторые его аспекты поощряются в культуре. Культура «компьютерщиков» всегда сторонилась идей приемлемого социального общения, а ее члены гордятся своим правом поступать так, как им хочется. Не поймите меня неверно — я всю жизнь была бунтарем. И все же в социальной жизни и умении общаться с другими на общепринятых условиях есть свои преимущества.

Социально непригодные компьютеры

Компьютеры, как и их создатели, известны своей социальной непригодностью. Тем не менее, в средах общения компьютеры начинают играть социальную роль или превращаются в посредников между людьми, участвующими в социальном общении. Однако их позиция в социальной жизни вовсе не делает технологию более социально доступной; функции компьютеров сильно зависят от того, что им разрешают делать люди. Таким образом, дело программиста — наделить технологию возможностями, которые позволят ей работать в социальной жизни.

Что это означает применительно к средам общения? Мы не понимаем, как на самом деле работает социальная жизнь. Поэтому мы строим грубые приближенные модели для нее и для человеческой психологии. В современном технологическом мире предпосылки для этих моделей часто делаются на основании материалов, почерпнутых из научной фантастики и популярной психологии, потому что мы гордимся своим полным непониманием социальной жизни. Упрощенное или механическое понимание социальной жизни характерно для личностей, склонных к аутизму.

С точки зрения аутиста явления социальной жизни могут и должны обрабатываться на программном и алгоритмическом уровне, и восприниматься на уровне простых категорий. Сложные отношения, в которые люди ежедневно вступают в повседневной жизни, сводятся к сегментированным возможностям. Когда мы учим детей, склонных к аутизму, участвовать в социальной жизни, мы объясняем им смысл таких простых понятий, как выражение лица: что улыбка означает доброжелательность, а нахмуренные брови — озабоченность. Шаг за шагом мы препарируем социальные аффекты и пытаемся формализовать их, чтобы дети

---

[1] В наши дни при проведении презентаций на компьютерных конференциях (вроде той, на которой делался этот доклад) часто бывает, что около 80 или 90 % аудитории пользуется портативными компьютерами. — *Примеч. ред.*

могли понять окружающий мир. То же самое мы пытаемся проделать с компьютерами. Как это отличается от простого вопроса: «Ты мне друг или нет?»

Возьмем недавний всплеск интереса к многозвенным социальным сетям — таким как Friendster, Tribe.net, LinkedIn, Orkut и т. д. Эти технологии пытаются формализовать процесс построения и управления отношениями среди людей и предполагают, что вы можете присваивать рейтинги своим друзьям. В некоторых случаях они на процедурном уровне управляют процессом общения с новыми людьми, предоставляя своим пользователям формализованный процесс для установления контактов.

Несомненно, такой подход обладает своими достоинствами, поскольку он легко формализуется, однако меня пугает, когда люди считают это моделью социальной жизни. Он настолько упрощен, что люди вынуждены вступать в общение так, словно они страдают аутизмом, и все взаимодействия между ними должны строиться на процедурном уровне. Конечно, процедурный подход помогает людям, нуждающимся в систематизации такого рода, но это не универсальная модель, которая подходит абсолютно всем. Более того, к каким последствиям приводит наличие технологий, требующих механистического вовлечения новых участников в процесс общения? Действительно ли нам нужна социальная жизнь, поощряющая аутистический стиль общения?

Всем известна склонность «технарей» строить программное обеспечение на основании своих собственных привычек и ценностей (вместо потребностей и ценностей других людей). Однако при таком подходе массовый пользователь часто остается в проигрыше; ему приходится либо принимать представления, выдвинутые разработчиками, либо отказаться от попыток. Если мы действительно пытаемся построить среду общения с поддержкой социальных взаимодействий, не следует ли взять за основу знакомые стереотипы социальной жизни? Почему бы не обеспечить поддержку громадного разнообразия нюансов, позволяющих людям общаться по-разному в зависимости от своих потребностей?

Ни одна из многозвенных социальных сетей не моделирует повседневную жизнь. Если вы захотите понять, в чем эти сети отклоняются от социальной жизни, и на какой теоретической основе они намеренно строились, обратитесь к другим моим работам[1]. Но поймите, что создание интегрированных систем подтверждения личности, обслуживающих эти сети, не решит фундаментальных проблем, заложенных в самой технологии. Невозможно лечить расстройство расщепления личности, чтобы решить проблему с аутизмом. А ведь именно это мы пытаемся сделать, говоря о протоколах FOAF.

Впрочем, это не означает, что упрощенные модели повседневной жизни неинтересны, и с ними нельзя поиграть. Люди любят видеть упрощенные модели самих себя. Как вы думаете, почему так популярны опросы типа «На какого персонажа "Звездных Войн" вы похожи?» Они не несут глубокого смысла, но предоставляют интересное отражение, возможность для социальной игры. Они открывают такие же возможности для внутреннего и внешнего обмена мнениями, как и карты таро, не претендуя на роль осмысленной модели чьей-либо социальной психологии.

[1] См. http://www.danah.org/papers

Упрощенные модели человеческого взаимодействия получили повсеместное распространение в нашей отрасли. В процессе быстрого принятия технологий, базирующихся на этих моделях, мы превозносим достоинства этих технологий, но не задумываемся, что же люди в действительности с ними делают.

Успех Friendster

Для примера возьмем Friendster. Этот сайт разрабатывался как своего рода служба знакомств. Предполагаемый сценарий использования был простым: дать людям возможность спланировать свою сеть общения, чтобы одинокие люди могли познакомиться с другими одинокими людьми в доверенной среде. И знаете, что произошло? Большинство пользователей отвергло этот сценарий. Даже те, кто пользовался функцией «представления», часто делал это для своих друзей, чтобы они могли подключиться к сайту. Успех Friendster не имел никакого отношения к его исходной цели или концепции. Вместо этого пользователи рассматривали Friendster как гибкую среду, которую можно было приспособить для отражения их собственных социальных привычек. Узнавая, как люди включают Friendster в свою повседневную жизнь, я был очарован тем, что так много разных людей находят ему так много разных применений. Одни воспринимали Friendster как инструмент для сбора информации о друзьях и посторонних. Он также использовался как игровая среда, канал распространения наркотиков, антидепрессант, конкурс популярности для будущих королев школьных балов, и т. д.

В течение некоторого времени было решено ограничить доступ к Friendster из-за вечно растущих простоев и неважного быстродействия. Одновременно прикладывались усилия к контролю над тем, что и как делали пользователи. Это отчасти сняло проблемы с нагрузкой за счет оттока ранних сторонников Friendster. Многим из них стало скучно, и они были разочарованы сайтом; он перестал обеспечивать весь диапазон интерактивных возможностей, которые привлекли их сюда. Тем не менее, появились другие группы пользователей с другими привычками, которые нашли новые механизмы взаимодействия с Friendster.

Простота Friendster позволяла снова и снова адаптировать его для новых целей. Популярность Friendster не доказала правильности базовой модели, многозвенных социальных сетей или ценностей, воплощенных в этой технологии. Успех Friendster доказал лишь то, что люди любят гибкие системы, которые позволяют им поразмышлять о своей личности и социальном окружении. Стремительный рост популярности Friendster объяснялся именно гибкостью, а не тем, что публика приняла ценности, заложенные в эту систему.

За последний год сотни компаний решили, что социальные сети — модная штука, и их необходимо внедрять повсюду. Мне часто говорят, что социальные сети — будущее общения в Интернете. Но на самом деле они всегда были краеугольным камнем Интернета. Отличается лишь то, что мы попытались механически упорядочить, формализовать их. Это вовсе не привело к неожиданному появлению социальных сетей как нового явления; формализация означала лишь то, что они стали менее серьезными, более «игрушечными». Все остальные социальные сети Интернета внедрены в другой комплекс стереотипов, не требующий отдельного практического применения, оправдывающего их существование.

В своей текущей версии социальные сети являются вспомогательным механизмом. Мы конструируем свою личность в контексте других людей. Мы приводим списки друзей и сообществ, чтобы показать, кто мы такие и во что верим. Мы заполняем свои блог-листы[1] списками людей, которыми мы восхищаемся. Эти сигналы многое говорят об индивиде, но ничего не говорят о его социальной сети — об отношениях доверия или потоке информации.

Люди часто спрашивают меня, куда ушли ранние поклонники Friendster. Конечно, некоторые из них перешли на Tribe.net, MySpace или в другие социальные сети, но подавляющее большинство вернулось к своей прежней жизни, в которой такие средства не используются. Они обратились к Friendster не ради возможностей социальных сетей, а потому, что Friendster хорошо вписывался в их стиль жизни.

Технологии и практическое применение

Меня попросили обсудить будущие перспективы, и я должна сказать, что я немного разочарована. Существует тенденция следовать за модой, совершенствовать модные штуки и исправлять технологические недостатки. Но я боюсь, что при этом мы теряем из виду «большую картину». В этом контексте имеет смысл не совершенствование технологии с тем, чтобы разобраться с социальными последствиями «когда-нибудь потом», или построение миллиона систем-клонов, будто копирование способно принести большие деньги. Вместо этого необходимо сделать шаг назад и подумать о том, какие социальные привычки мы намерены отразить, и какие ценности мы закладываем в этих попытках. Самыми успешными социальными технологиями являются те, которые укладываются в стиль жизни и привычки людей; они заполняют существующие пробелы вместо того, чтобы создавать искусственные потребности. Даже LiveJournal базируется на стандартной привычке: ведении дневника. Эта система тоже развивалась на базе того, что пользователи LiveJournal делали в сети и в реальном мире, и существующая практика уже мало напоминает ведение дневника.

Другими словами, я не хочу сказать, будто мы можем предсказать, какие технологии хорошо войдут в людскую жизнь — однако мы можем учиться на уроках существующих технологий при создании наших собственных разработок.

Чтобы заставить технологию работать в контексте людских потребностей, возможны три пути:

1. Разработать технологию, разрекламировать ее до небес и потребовать, чтобы она вписалась в жизнь людей. Когда из этого ничего не выйдет — использовать метод «нагрузки». Иначе говоря, упакуйте свою разработку во что-нибудь нужное, чтобы люди были вынуждены ей пользоваться. Лично мне этот вариант кажется отвратным, хотя я понимаю, что так работает большая часть нашей отрасли.

---

[1] Список других блогов, размещаемый на полях вашего блога; обычно включает часто посещаемые вами блоги, блоги ваших друзей или те блоги, которые, как вы надеетесь, разместят обратную ссылку на ваш блог. — *Примеч. ред.*

2. Создать технологию, выложить ее в открытый доступ и посмотреть, что приживется. Последовать за людьми, которые ей пользуются. Понять их. Понять, что они делают, зачем, и какое место во всем этом занимает ваша технология. Развить технологию, чтобы она лучше отвечала потребностям и желаниям людей, которым она пришлась по душе.
3. Понять группу людей и их потребности, а затем разработать технологию, которая естественным образом укладывается в привычки этих людей. Сделать технологию общедоступной.

Лично я считаю, что два последних способа представляют добросовестный подход к проектированию социальных технологий. Третий способ — распространенный механизм, используемый аналитиками в этой области, а второй обещает стать полезным, когда проектировщики социального программного обеспечения остановятся и посмотрят, что же у них получилось, вместо того, чтобы выпустить новую технологию для решения технологических проблем.

Мы говорим о технологиях, предназначенных для общения людей с другими людьми. Пользователи могут вытворять самые невероятные вещи, но они выглядят странно только тогда, когда вы пытаетесь понять их со своей точки зрения. Попробуйте взглянуть на происходящее с точки зрения тех же пользователей. Вместо того, чтобы требовать от них нужного вам поведения, попробуйте понять, почему они выбирают путь, отличный от вашего. Когда вы начнете понимать логику их поступков, задача наполовину решена.

Самое сложное состоит в том, чтобы проектировать с точки зрения пользователя; полностью воспринять ее, а не просто допускать ее существование; формировать технологию на основании этой точки зрения, потому что именно пользователи способствуют эволюции. Когда мы ворчим по поводу этих противных пользователей, мы упускаем самое важное. Пользователи взаимодействуют с технологией не для того, чтобы что-то доказать нам. Они делают это, потому что технология укладывается в их представления о мире. Понимание, истинное понимание... вот ключ к успеху технологии.

В завершение мне хотелось бы привести цитату из статьи Дугласа Адамса «Как я перестал беспокоиться и полюбил Интернет»:

> «Очень большая часть нашего мозга развивалась именно для выработки социальных принципов относительно того, кому можно доверять и почему».

Социальное программное обеспечение обладает потенциалом для настоящего реформирования процесса разработки технологий. Новые фирмы бросают технологии в массы, и массы работают с ними, находя непредвиденные новые применения. Мы можем либо повернуться к ним спиной и выпрашивать деньги у инвесторов, используя нашу собственную точку зрения, либо учитывать их интересы и убедить мир в том, что это более сознательный и ценный в долгосрочной перспективе подход для всех участников. Я отдаю свой голос за то, чтобы сосредоточиться на интересах обычных людей и перестать требовать, чтобы они участвовали в наших аутистских затеях. Давайте предоставим им возможность осмысленно использовать подходы к социальной жизни во всех тонкостях.

[Спасибо Кори Доктороу (Cory Doctorow), Скотту Ледереру (Scott Lederer), Кевину Марксу (Kevin Marks) и многим другим за хорошие идеи и общение.]

Рэймонд Чен

Почему бы просто не заблокировать приложения, зависящие от недокументированных функций системы?

Windows содержит целый ряд нетривиальных и изящных фокусов, которые гарантируют, что при переходе на новую версию операционной системы все старые приложения продолжат работать.

Я впервые узнал об этом от одного из разработчиков популярной игры SimCity. Он рассказал мне, что в приложении была допущена критическая ошибка: оно использовало память сразу же после ее освобождения. Это серьезное нарушение, которое нормально работало в DOS, но отказывалось работать в Windows, где освобожденная память с большой вероятностью немедленно захватывалась другим выполняемым приложением. Группа тестирования проверяла различные популярные приложения и правильность их работы в Windows, но игра SimCity раз за разом зависала. Об ошибке сообщили разработчикам Windows; те дизассемблировали SimCity, выполнили пошаговую отладку, нашли ошибку и включили в Windows специальный код. При запуске SimCity этот код переводил распределитель памяти в специальный режим, в котором было возможно использование памяти после ее освобождения.

Молодые и неопытные программисты от подобных вещей лезут на стены. Однако именно такие трюки определили успех Windows. Рэймонд Чен, ветеран из группы Windows компании Microsoft, сможет объяснить это лучше других. — *Ред.*

Почему бы просто не заблокировать приложения, зависящие от недокументированных функций системы? Потому что каждое заблокированное приложение станет лишним поводом для отказа от перехода на следующую версию операционной системы Microsoft® Windows®. Только посмотрите[1], сколько программ перестало бы работать при переходе с Windows 3.0 на Windows 3.1:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\
Windows NT\CurrentVersion\Compatibility
```

Причем этот список далеко не полон. Во многих случаях решение проблемы совместимости встраивается в базовые компоненты сразу для всех программ, а не для одной конкретной программы, как в этом списке.

---

[1] В системном реестре; запустите regedit. — *Примеч. ред.*

(Аналогичный список для перехода с Windows 2000/Windows XP хранится в каталоге C:\WINDOWS\AppPatch в двоичном формате, повышающем скорость сканирования. Для его просмотра следует использовать программу Compatibility Administrator из пакета Application Compatibility Toolkit[1]).

Стали бы вы покупать Windows XP, зная о несовместимости всех этих программ?

Всего одна несовместимая программа может повлиять на решение об обновлении.

Допустим, вы руководите технической службой некоторой компании. В вашей компании используется редактор ProgramX; выясняется, что программа ProgramX по какой-то причине несовместима с Windows XP. Станете вы обновлять систему?

Конечно, нет! Ведь это парализует всю работу.

«Почему бы не позвонить в компанию CompanyX и не попросить обновление?»

Разумеется, это можно сделать. Вам ответят: «А, так вы используете ProgramX версии 1.0. Вам нужно обновить ее до версии 2.0, это будет стоить $150 за одну копию». Поздравляю, затраты на переход к Windows XP возрастают втрое.

Притом это только в том случае, если вам повезло, и CompanyX еще продолжает работу над продуктом.

Припоминаю результаты одного опроса, проведенного нашей группой обновления среди корпораций, использующих Windows. Практически в каждом случае отыскивалась одна «критическая» программа, которая в обязательном порядке должна была поддерживаться Windows, иначе компания отказывалась от обновления. Нередко программа была написана кем-то из программистов самой компании на Microsoft Visual Basic® (иногда даже на 16-разрядном Visual Basic), а человек, который написал программу, давно уволился. В отдельных случаях не было даже исходного кода программы.

Причем проблема касается не только корпоративных клиентов, но и рядовых пользователей.

В Windows 95 моя работа по проверке совместимости приложений была направлена в основном на игры. Игры давно стали важнейшим фактором потребительских технологий. Видеокарта типичного компьютера со временем становится все более мощной, потому что этого требуют компьютерные игры (конечно, Microsoft Office Outlook® совершенно не нужно, чтобы ваша видеокарта умела рисовать 20 суперпупериллионов треугольников в секунду). А если ваша игра откажется работать в новейшей версии Windows, вы откажетесь от обновления.

Как бы то ни было, разработчики игр в чем-то очень похожи на крупные корпорации, о которых я говорил. Я обзванивал одну компанию за другой, пытаясь добиться от них помощи в запуске игры под Windows 95. Их это совершенно не волновало. Жизненный цикл игры длится несколько месяцев, и он уже завершился. Зачем тратить время на выпуск заплатки, обеспечивающей запуск игры в Windows 95? Деньги уже получены. На этой игре больше не заработаешь; ее время прошло уже три месяца назад. Разработчики неохотно идут на выпуск заплаток; они перестают следить за тем, какие версии программы были выпущены,

[1] См. http://www.microsoft.com/windows/appcompatibility/.

и в каких версиях существовала та или иная проблема. Иногда у них даже не остается исходных кодов программы.

Короче, их попросту не волновало, что программа не запускается в Windows 95.

(Мой любимый пример — программа, которая пыталась руководить моими действиями при создании загрузочной дискеты).

Да, и пакет Application Compatibility Toolkit, о котором я уже упоминал, — этот замечательный инструмент был крайне полезен и для разработчиков. Одним из его компонентов был Windows Application Verifier: если запустить вашу программу под его управлением, он отслеживал вызовы сотен функций API[1] и передавал управление отладчику при выполнении недопустимых действий (скажем, повторного закрытия идентификатора (handle) или выделения памяти функцией GlobalAlloc с последующим освобождением функцией LocalAlloc).

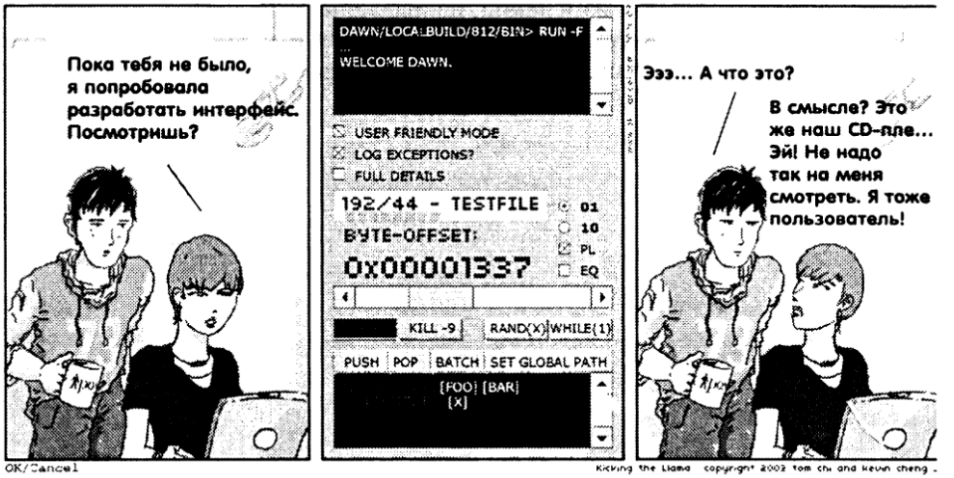
Новая архитектура совместимости приложений в Windows XP обладает одним серьезным преимуществом (с точки зрения разработки ОС): видите длинный список DLL-библиотек в каталоге C:\WINDOWS\AppPatch? Именно здесь теперь сосредоточено большинство изменений, направленных на сохранение совместимости. Заплатки совместимости перестают загрязнять основные файлы ОС (хотя не все решения проблем совместимости удается оформить в виде DLL, этот механизм все равно чрезвычайно полезен)[2].

[1] Функция API — запрос от программы к системе Windows на выполнение какой-то операции. — *Примеч. ред.*

[2] DLL — файл с расширением .DLL, содержащий код, который может использоваться уже работающей программой. Например, если вы хотите написать программу проверки орфографии, которая могла бы использоваться всеми программами, но не занимала память во время бездействия, оформите ее в виде DLL и загружайте из своих редакторов, электронных таблиц и т. д. по мере надобности. Большая часть функциональности Windows реализована в формате DLL, и в каталоге Windows можно найти сотни этих библиотек. — *Примеч. ред.*

Кевин Ченг и Том Чи

Пинок для ламы[1]



Намек на пользовательский интерфейс WinAmp: как известно, талисман этой программы — лама, а девиз — «it really whips the llama's ass». — *Примеч. перев.*

Кори Доктороу

Спасите канадский Интернет от WIPO

Знаю отличный фокус. Попробуйте сами.

Выберите какого-нибудь ненавистного вам субъекта. Неважно, за что вы его ненавидите.

Найдите его сайт.

Выберите случайный абзац на этом сайте.

Отправьте провайдеру сообщение с жалобой на то, что этот абзац является нарушением авторского права, и апеллируйте к DMCA. Вам даже не нужно знать, что означает это сокращение. И конечно, абзац совсем не обязан что-либо нарушать. Просто скажите, что авторские права принадлежат вам, а абзац был без разрешения скопирован из вашего школьного реферата. В 9 случаях из 10 ничего больше не потребуется. Провайдер выкидывает вашего врага из своих учетных данных. Та-да! Правосудие быстрое и беспощадное.

Много, много лет назад, когда телефонная система еще была правительственной монополией, владельцы авторских прав попытались судиться с Bell, когда телефонная система использовалась для пересылки защищенных материалов. Почему они подавали в суд на телефонную компанию, а не на человека, который пересылал защищенные материалы? По той же причине, по которой Вилли Саттон грабил банки: там были деньги.

Естественно, телефонные компании не несли ответственности за нарушения и не могли выполнять функции «полиции по авторскому праву» для всех телефонных звонков; вскоре закон согласился с этим. В тех случаях, когда некая сторона выполняет функции простой передачи информации, непредвзято пересылая все данные независимо от их содержания, она не может считаться ответственной за юридические аспекты этих данных.

В раннюю эпоху Интернета тот же принцип был позаимствован из области телефонии. Если вы нанимаете меня доставить данные из точки А в точку Б, совершенно неважно, что представляют собой эти данные — случайный набор битов, защищенный материал, требования выкупа, любовные письма или видеоклипы; ответственность за содержимое несу не я, а вы. Я всего лишь доставляю данные. Это единственный юридический принцип, отвечающий здравому смыслу.

К сожалению, здравый смысл не подходил боссам музыкальной отрасли. От одной мысли о тех суммах, которые им не удавалось переложить из карманов

исполнителей в собственные карманы из-за обмена mp3-файлами, они находились на грани припадка. Они хотели больше прав для борьбы с пиратством, и они хотели иметь возможность преследовать организации, нанятые для пересылки данных из точки А в точку Б — потому что, как сказал Вилли Саттон, там были деньги. Поэтому в результате долгого и усердного лоббирования был принят акт DMCA — коварный закон, который добавлял одно внешне невинное положение: пересылая данные из точки А в точку Б, вы не несете ответственности за их содержание, но только в том случае, если вы согласитесь снять весь материал, на который кто-нибудь пожалуется.

Неважно, что это выглядит полным бредом; неважно, что у каждого злобного идиота появляется практическая возможность подвергнуть цензуре любую информацию на сайте, отправив всего одно письмо. Неважно, обоснована жалоба или нет; обычно провайдер предпочитает подчиниться цензорскому запросу без каких-либо расследований.

Закон плохой, но в этом виноваты мы сами, разрешая лоббистам индустрии развлечений писать наши законы. — *Ред.*

Канада серьезно рассматривает возможность ратификации «Интернет-соглашений» WIPO 1996. Именно эти соглашения заставили Соединенные Штаты принять отвратительный акт DMCA (Digital Millennium Copyright Act, «Акт об авторском праве в цифровом тысячелетии»), причинивший огромный ущерб по всему миру. Что это будет означать для Канады? Для начала *Globe and Mail* отмечает, что режим «оповещения со снятием» неизбежен:

«...В вопросе, по которому неизбежно возникнут разногласия, комитет порекомендовал, чтобы провайдеры Интернета несли ответственность за нарушения авторского права в материалах, проходящих через их системы. Чтобы освободиться от этой ответственности, провайдер должен показать, что он выполняет функции посредника без фактического или конструктивного знания содержания.

Провайдер должен применять систему "оповещения со снятием" против подписчиков, нарушающих законы авторского права».

Несравненный сисадмин Boing Boing — канадский хакер Кен Снайдер (Ken Snider) — написал:

«Очень важно, чтобы наше правительство не склонилось перед CIRA[1]. Я питаю большие надежды, что нынешнее правительство меньшинства не станет решать эту проблему в обозримом будущем, но я *хочу* донести свое сообщение до каждого чертова члена парламента, до которого я смогу дотянуться. Проблема в том, что у меня нет никакой *конкретной* информации по этим законопроектам. Я надеялся, что ты хотя бы направишь меня в нужном направлении (или даже выступишь рядом со мной! Ууу!) Считаю *критически* важным, чтобы Канада не последовала за США в этом процессе. Я готов сделать все возможное, чтобы предать огласке причины того, *почему* эта идея так плоха. Просто мне нужна небольшая помощь для четкого и лаконичного представления своих доводов, а также наполнения их достаточным количеством "политики", чтобы на них обратили внимание».

[1] Вероятно, Canadian Internet Registration Authority. — *Примеч. перев.*

Итак, Кен, вот кое-какая информация для тебя. В мире полно регулирующих технологий: одни правила управляют применением автомашин, другие — применением электроприборов, и т. д. В правилах как таковых нет ничего плохого.

Но когда был изобретен локомотив мощностью 20 лошадиных сил, кузнецам так и не удалось успешно провести закон, заставляющий приваривать к каждому двигателю 80 подков — хотя правило гласило, что каждая «лошадь», используемая как транспортное средство, должна иметь четыре подковы. Если вы изобретаете железную дорогу, для нее необходимо писать железнодорожные правила, не ограничиваясь правилами для лошади с телегой. Тот факт, что паровозам не нужны подковы, овес или скребки, не является недостатком железных дорог; это особое свойство технологии.

У Интернета есть одно основополагающее свойство, определяющее его превосходство над всеми предшествующими технологиями: он позволяет копировать произвольные блоки данных из одного места в другое практически без затрат, практически моментально и практически бесконтрольно. Это не дефект. Именно так и должен работать Интернет.

В 1996 г. WIPO[1] весьма дальновидно занялась обновлением законодательства об авторском праве для Интернета. В WIPO поняли, что Интернет принципиально отличается от всех ранее существовавших технологий, и конечно, для новой технологии нужны новые правила и уставы. Однако в WIPO допустили чудовищную ошибку. Подход, выбранный для регулирования Сети, был основан на создании набора правил, которые пытались заставить Интернет работать по тем же принципам, что и радио, телевизор или ксерокс — как все те вещи, для которых уже были созданы свои правила. В WIPO простота копирования данных в Сети рассматривалась как дефект, который было решено исправить.

«Оповещение со снятием» — та самая область, в которой WIPO допустила свою огромную, ужасную ошибку.

Владелец ресторана не отвечает за то, чтобы его клиенты не носили ворованного добра. Если кто-то вломится в ваш ресторан, укажет на субъекта за столом и скажет: «На нем моя шляпа!», никто не заставит вас отнимать шляпу и отдавать ее обвинителю. Но ведь именно этого мы требуем от провайдеров Интернета. Разрешая пользователю размещать данные, вы несете за них ответственность. Если на сервере размещается файл, нарушающий законодательство, вы обязаны знать об этом, а если вы не помешали публикации таких материалов — то становитесь соучастником преступления.

По поводу того, что должно считаться нарушением авторского права, тоже нет особой ясности. Провайдер не располагает средствами, чтобы понять, где есть нарушения, а где их нет — даже судьям Верховного Суда бывает нелегко в этом разобраться. Управление сервером еще дает необходимой квалификации для понимания и оценки нарушений авторского права.

И здесь возникает индульгенция в виде «оповещения со снятием». Если вы отреагируете на обвинения в нарушении авторского права, быстро перекрыв

---

[1] World Intellectual Property Organization (Всемирная организация по охране интеллектуальной собственности, ВОИС). — *Примеч. перев.*

доступ к материалам своего клиента, то не разделите ответственность вместе с ним. А если учесть, какие наказания могут применяться к нарушителям авторских прав (в США до $150 000 за нарушение!), вполне понятно, что большинство провайдеров предпочитают «действовать наверняка» и удаляют материалы при любом поступлении жалобы.

Однако жалоба еще не является доказательством — тот, кто обращается к провайдеру и говорит: «Этот файл нарушает мои права», напоминает того субъекта, который врывается в ресторан и кричит: «На нем моя шляпа!». Провайдер не может сходу определить, действительно ли он является пострадавшим, решил кому-то отомстить, или это обычный псих. В США любой псих или мстительный пакостник получает возможность вводить цензуру в Интернете и снимать материалы по своему усмотрению.

Обычно обладатели авторских прав указывают на то, что для решения подобных проблем существует механизм «встречного оповещения»: провайдер, снимающий материалы, имеет право обратиться к клиенту и сказать: «Этот человек говорит, что вы нарушили его авторское право. Если вы не согласны, сообщите нам — мы вернем ваш файл на место, а вы двое сможете решить свои проблемы в суде». Но на практике встречные оповещения встречаются крайне редко. Как правило, провайдер прикидывает и решает, что одно письмо с встречным оповещением может им стоить больше адвокаточасов, чем они заработают на этом клиенте. Тогда они инициируют статью о досрочной отмене, присутствующую практически в каждом контракте с провайдером, и закрывают учетную запись клиента.

Вот почему оповещение со снятием является почти идеальным инструментом цензуры. Не нравится, что говорят ваши критики? Отправьте оповещение, и неприятная информация исчезнет! Сайентологи обожают эту тактику — они даже заставляют Google удалять ссылки на сайты с критикой их «церкви», заявляя о нарушениях авторского права. Только просмотрите хроники сайта ChillingEffects[1], на котором собраны оповещения о нарушении авторского права и прочие гадости. Оповещение о нарушении авторского права становится любимым инструментом психопатов, цензоров и наглецов.

Даже применительно к настоящим нарушениям авторских прав система снятия опасна до уровня практической непригодности: Business Software Alliance, MPAA и RIAA тысячами рассылают автоматически сгенерированные оповещения, используя убогие программы поиска по шаблону среди файлов, доступных в Сети. Затем среди университетов, провайдеров и других организаций рассылаются письма с требованиями снять рецензии на книги о Гарри Поттере, дистрибутивы Linux, названия которых совпадают с названиями фильмов, а также академические труды профессоров, являющихся однофамильцами известных музыкантов. Более того, оповещение со снятием почти всегда сопровождается нарушением анонимности пользователей Интернета: если вы захотите узнать новый адрес и номер своей жертвы, достаточно найти форум, где она сообщает о своих бедах. Дальше остается только написать провайдеру от имени владельца нарушенных прав и затребовать соответствующие данные.

---

[1] См. http://chillingeffects.org.

Если Канада захочет «решить» проблемы Интернета, ей следует искать решения, учитывающие специфику Интернета. Интернет-законодательство Канады должно рассматривать копирование как особенность, а не как дефект. Оно должно эмпирически оценить, на какие секторы окажет отрицательное влияние обмен файлами (все больше доказательств свидетельствует о том, что Интернет практически никогда не виноват в трудностях индустрии развлечений), а затем решить проблемы этих отраслей с помощью генеральных лицензий и других средств, не пытающихся ограничить копирование — это невозможно сделать без нарушения работы Интернета. Решения, которые изначально рассматривают Интернет как проблему, ничего решить не могут.

ea_spouse

EA: житейская история

В области разработки программного обеспечения существует крупное течение, к которому я принадлежу: его сторонники считают, что программисты, подолгу работающие более 40 часов в неделю, выполняют меньше полезной работы, чем программисты, которые обходятся без хронических авралов. Это явление было хорошо изучено и описано в таких книгах, как «Peopleware»[1] Тимоти Листера и Тома ДеМарко. Если человек занимается программированием более восьми часов в день, качество его работы снижается настолько, что на каждый час программирования приходится тратить два часа на исправление ошибок. Работа, выполняемая после восьми часов, не приносит желаемых результатов. Также я глубоко уверен в том, что опытный работник гораздо ценнее новичка, и для того, чтобы новичок смог набрать полный темп и трудиться так же продуктивно, как более опытные коллеги, может потребоваться целый год. Если ea_spouse верно утверждает, что текучесть кадров в EA составляет 50 % в год из-за сверхурочной работы, у этой компании серьезные проблемы.

Ea_spouse рассматривает ситуацию с точки зрения обычных людей — работников и их семей. Такая точка зрения тоже очень важна.

Но даже если смотреть на происходящее только с точки зрения работодателя, если думать лишь о том, чтобы компания Electronic Arts добивалась максимальной прибыли, политика постоянного аврала совершенно не оправдывает себя.

Давайте немного посчитаем.

Если заставлять программистов работать по 90 часов в неделю, у них не остается времени на мелкие дела, являющиеся частью повседневной жизни. Если программист работает с 9 утра до 10 вечера по 7 дней в неделю, когда он должен ставить свою машину на техосмотр? Когда он должен оплачивать счета? Или звонить маме? Я скажу: когда он находится на работе. Все это происходит во время работы, поэтому из реальной рабочей недели следует сходу вычесть 10 часов производительной работы. Ладно, остается 80.

Много ли пользы от 40 сверхурочных часов? Скорее всего никакой. Большинство программистов, когда их заставляют сидеть на работе допоздна, используют

---

[1] DeMarco, Tom and Timothy Lister. Peopleware: Productive Projects and Teams, 2nd Ed. Dorset House, 1999.

дополнительное время на путешествие в Web, болтовню в чатах и на все что угодно, кроме программирования — и не из-за патологической лени, а потому, что их мозг отработал дневную норму. Но давайте предположим, что хотя все доказательства свидетельствуют об обратном. за эти лишние 40 часов все же выполняется 10 часов полезной работы. Остается 50 полезных часов.

Теперь вычтем затраты на наем новых работников в замену «перегоревших». Наем и обучение нового работника обычно оценивается в сумму, эквивалентную годовому окладу. Сюда включены не только непосредственные затраты на наем, но и низкая производительность нового работника на то время, пока он входит в курс дела и впитывает информацию от других работников, расходы на переезд, аванс на обустройство и т. д.

Если EA ежегодно теряет 50 % своих работников (тогда как стандартный показатель по отрасли составляет около 5 %), это эквивалентно расширению штата компании на 45 %. Или если выразить происходящее в контексте 50-часовой рабочей недели, фактическая продолжительность рабочей недели среднего работника составляет немногим более 25 часов, потому что почти половина персонала работает первый год и еще не отработала своих начальных затрат. Получается, что 40-часовая рабочая неделя не только более гуманна, но и значительно более выгодна для Electronic Arts. Я говорю это не по каким-то извращенным идеологическим соображениям: у меня собственная компания по разработке программного обеспечения, в которой строго соблюдается правило 40-часовой рабочей недели, так что мои слова подтверждаются практикой.

В марте 2005 года компания Electronic Arts объявила, что она «отходит от традиции и начинает платить сверхурочные некоторым работникам. Такие работники теряют право на получение льгот и премий». Хм... Что же, с чего-то нужно начинать. Но, похоже, EA делает минимум того, что необходимо для соблюдения законов, не понимая в полной мере, насколько непродуктивна потогонная система в программировании. — *Ред.*

Мой супруг работает на Electronic Arts, а я из тех, кого обычно называют «рассерженной половиной».

Эффектный новый корпоративный девиз EA гласит: «Испытай себя во всем». Мне не совсем понятно, о чем идет речь. На мой взгляд, клепать по лицензии одну за другой футбольные игры — не такое уж большое испытание; скорее напоминает выкачивание денег. Могу предложить хорошее испытание любому из руководителей EA, которому доведется это прочитать: как насчет безопасных и вменяемых условий труда для людей, по спинам которых вы идете к своим миллионам?

Я предпочитаю сохранить анонимность — у меня нет иллюзий относительно того, к каким последствиям для моей семьи приведет моя откровенность. Тем не менее, у меня также нет причин скрывать нашу историю, потому что она слишком обычна и нисколько не выделяется на фоне историй тысяч программистов, художников и дизайнеров, работающих на EA.

Наши приключения с Electronic Arts начались меньше года назад. Маленькая игровая студия, на которую работал мой партнер, развалилась в результате грязной игры со стороны крупной фирмы — вполне обычное дело. Компания Electronic Arts предложила работу с нормальным окладом и хорошими льготами, поэтому мой супруг согласился. Помню, в одном из собеседований его спросили: «Как вы относитесь к сверхурочной работе?» Это часть игровой индустрии — мало кому

из студий удается избежать авралов с приближением конечных сроков, и мы ничего не заподозрили. На вопрос, что именно означает «сверхурочная работа», собеседники закашлялись и перешли к следующему вопросу; теперь мы знаем, почему.

За несколько недель темп работы повысился до «умеренного аврала»: восемь часов, шесть дней в неделю. Ничего страшного. До начала настоящего аврала оставались месяцы, и группе сообщили, что небольшая переработка сейчас предотвратит большой аврал в конце; в то время любые другие доводы для аврала выглядели маловероятными, поскольку работа шла идеально по расписанию. Не знаю, сколько разработчиков поверило объяснениям EA по поводу сверхурочных; мы были неопытны и наивны, поэтому тоже поверили. Руководители даже установили предельный срок; они назначили конкретную дату окончания аврала, которая на несколько месяцев отстояла от даты выхода проекта, так что все выглядело вполне благопристойно. Назначенный день наступил и прошел. А потом еще, и еще, и еще... Когда пришли следующие новости, они несли не передышку, а новое ускорение: 12 часов, 6 дней в неделю, с 9 утра до 10 вечера.

Проходили недели. Снова руководство назначило дату завершения аврала, и снова не сдержало своих обещаний. В течение этого периода проект шел по графику. Усталость понемногу начала сказываться на людях; они стали раздражительными, а некоторые начали болеть. Начались массовые отлучки на день-другой, но потом группа снова достигла равновесия и принялась пахать. Начальство перестало даже упоминать о дне, когда рабочее время вернется в норму.

Похоже, настал «настоящий» аврал, к которому руководство проекта так «мудро» подготовило свой персонал, загоняв его раньше времени. Обязательный рабочий день продолжался с 9 до 22, семь дней в неделю, хотя иногда за хорошее поведение по субботам отпускали пораньше (с 18:30). Средняя продолжительность рабочей недели составляла 85 часов. Жалобы на то, что сверхурочная работа в сочетании с усталостью группы лишь повышает количество ошибок — и оборачивается еще большими напрасными потерями — игнорировались. Напряжение не проходило бесследно. После нескольких часов работы начинали болеть глаза, а после нескольких недель с одним выходным усталость накапливалась по экспоненте. Именно по этой причине в конце недели даются два выходных — если лишить работника этих дней, это приведет к печальным последствиям для его физического, эмоционального и душевного здоровья. Группа вскоре начинает делать ошибки с такой же частотой, с какой они исправляются.

А теперь самое невероятное: за такое обращение работники не получали: 1) сверхурочных; 2) временных компенсаций! (когда лишние часы, потраченные во время авралов, преобразуются в выходные после завершения продукта); 3) дополнительных отпусков или отлучек по болезни. Время просто пропадает. Кроме того, в EA недавно объявили, что хотя в прошлом предоставлялась некая форма временных компенсаций в виде нескольких недель отпуска после завершения проекта, больше такого не будет, и работникам не стоит на это рассчитывать. Более того, из-за разброса в сроках ведения проектов многие работники беспокоились, что они выйдут из одного аврала только для того, чтобы сходу попасть в другой. В EA ответили, что они постараются свести вероятность подобных событий к минимуму, но обещать ничего не могут. Это немыслимо; фактически доводя группу до грани физического срыва, они ничего не давали

взамен. Временные компенсации — обычное явление в этой отрасли, но EA как корпорация желает «свести к минимуму» эту передышку. Вообще говоря, правильным способом сведения к минимуму временных компенсаций был бы отказ от авралов, но эта жестокая гонка продолжалась месяцами без малейших намеков на отпуск, не говоря уже о ее прекращении.

Этот аврал также отличался от авралов в мелких студиях тем, что он не был экстренной мерой по спасению проекта. На каждой стадии проект оставался в рамках графика. Аврал не ускорял и не замедлял работы; он никак не влиял на проект. Сверхурочная работа была намеренной и запланированной; руководство отлично понимало, что оно делает. Любовь моей жизни возвращается поздно ночью, жалуясь на непроходящую головную боль и хроническое расстройство желудка, а моя радостная сочувствующая улыбка постепенно исчезает.

В игровой индустрии работают только те люди, которые искренне увлечены своим делом. Ни один участник группы не заинтересован в ухудшении качества продукта. Моя душа болит за эту группу именно потому, что она состоит из выдающихся, талантливых личностей, стремящихся создать нечто замечательное. И прежде, и сейчас они не боялись усердно трудиться ради успеха продукта. Но эта добрая воля лишь становилась предметом злоупотребления. Как ни удивительно, в 2003 году компания Electronic Arts находилась на 91 месте в списке «100 лучших компаний для работы» журнала Fortune.

Отношение EA к этой проблеме (которое, как оказалось, является частью политики компании) было сформулировано в анонимном высказывании, которое мне доводилось слышать от разных начальников: «Если кому-то не нравится, он может работать в другом месте». Либо смирись, либо заткнись или уходи: в этом заключается суть кадровой политики EA. Концепциям этики, сочувствия или хотя бы разумности столь жесткой эксплуатации рабочей силы здесь нет места: если они не хотят жертвовать своими жизнями, здоровьем и талантом, чтобы многомиллиардная корпорация продолжала поступью Годзиллы шагать по игровой индустрии, они могут работать в другом месте.

Но так ли это?

EA Mambo в обществе других гигантов вроде Vivendi, Sony и Microsoft, быстрыми темпами либо сокрушает, либо поглощает подавляющее большинство компаний, занимающихся разработкой игр. Нескольким отдельным студиям, заработавшим свои капиталы в более раннюю эпоху — на ум приходят Blizzard, Bioware и Id — удалось выжить, но 2004 год ознаменовался крахом десятков мелких игровых студий, не способных получить контракты перед лицом быстрой и массивной консолидации издателей игр. Вряд ли в отрасли найдется хоть один человек, незнакомый с этой эпидемией — хотя конечно, талантливый человек всегда может уйти из отрасли...и например, направить усилия в процветающую область разработки коммерческого программного обеспечения (усталые потуги на сарказм).

Чтобы представить происходящее в перспективе, я приведу несколько цифр. Если EA действительно считает, что столь жесткая эксплуатация работников необходима — хотя на самом деле я думаю, что авралы объясняются только искаженными представлениями о методах работы в сочетании с определенной изначальной неэффективностью, связанной с управлением столь крупной компанией — проблему следовало бы решать иначе: привлечением дополнительных

программистов, художников или дизайнеров. Ни при каких условиях эта мера не должна заменяться пыткой персонала 90-часовыми рабочими неделями; в любой другой отрасли компанию засудили бы так быстро, что ее акции даже не успели бы обвалиться. За первые выходные игра «Madden 2005» заработала $65 млн. Годовой доход EA составляет приблизительно $2,5 млрд. Компания не нуждается в жесткой экономии; ее трудовая политика непростительна.

Самое интересное во всем этом — это предположение, которым, похоже, руководствуется большинство работников. Каждый раз, когда речь заходит о продолжительности рабочего времени, кто-то неизбежно упоминает об «освобождении». Имеется в виду закон штата Калифорния, который предположительно освобождает коммерческие организации от выплаты сверхурочных отдельным категориям работников, включая программистов. Это Сенатский билль 88. Тем не менее, Сенатский Билль 88 не относится к индустрии развлечений — в тексте особо упомянуты телевидение, кино и театры. Более того, даже в области программного обеспечения установлен минимальный уровень оклада: лица, лишаемые выплат, должны зарабатывать не менее $90 000 в год. Уверяю вас, что большинство работников EA не входит в эту группу; следовательно, поведение EA не только неэтично, но и незаконно.

Я смотрю на ситуацию и спрашиваю «нас»: почему же вы остаетесь? Ответ: скорее всего, мы не останемся; и скорее всего, если бы мы знали, к чему приведет работа на EA, мы бы держались подальше от этой компании. Но мы столкнулись с измышлениями, обещаниями, заверениями; видели большое, эффектное офисное здание с дорогим аквариумом — все это, в конечном счете, обернулось хорошо продуманной системой, в которой работники удерживаются ровно столько, сколько необходимо для успешной сдачи проекта. А потом при необходимости нанимается новый коллектив, свежий и готовый выслушать новые обещания, которые никто не собирается выполнять; текучесть кадров среди программистов в EA составляет приблизительно 50 %. Так работает EA. Но теперь, когда мы все знаем, можно уходить, верно? Похоже, это происходит со всеми, но этого недостаточно. В конечном счете, чем бы ни закончилась наша конкретная ситуация, подобный «бизнес» недопустим, и люди должны знать о нем. Вот почему я пишу это сегодня.

Если бы генеральному директору EA Ларри Пробсту (Larry Probst) можно было позвонить по телефону, мне хотелось бы спросить у него несколько вещей. Вопрос «Какой у вас оклад?» — не более чем дань любопытству. Главное, что мне хотелось бы знать, Ларри: ведь вы понимаете, что делаете с людьми, верно? И вы понимаете, что это люди, у которых есть свои пределы физических возможностей, эмоции и семьи? Есть право голоса, талант, чувство юмора и все такое? И когда вы держите наших мужей, жен и детей в офисе по 90 часов в неделю, когда они возвращаются усталыми, оцепеневшими и недовольными жизнью, вы вредите не только им, но и всем окружающим, всем, кто их любит? Рассчитывая прибыли и проводя анализ затрат, вы знаете, что большая часть этих затрат покрывается простым человеческим достоинством, верно?

Верно?

Брюс Эккель

Сильная типизация против сильного тестирования

Помню, когда я работал над VBA в Microsoft, мы подолгу спорили по поводу статической и динамической проверки типов.

Термин «статическая проверка типов» означает, что компилятор на стадии компиляции убеждается в том, что все переменные относятся к правильному типу. Например, если в программе имеется функция log(), которой должно передаваться число, и в программе встречается вызов log("foo") с передачей строки, при статической проверке типов компилятор скажет: «Постой-ка! Этой функции нельзя передавать строку, потому что она ожидает число», и программа не будет компилироваться.

С другой стороны, динамическая проверка типов осуществляется на стадии выполнения. При динамической проверке типов вызов log("foo") откомпилируется нормально, но во время выполнения программы произойдет ошибка. Недостаток этого способа состоит в том, что ошибка может остаться незамеченной до того момента, когда через несколько месяцев кто-то не выполнит эту строку кода (особенно если она находится в редко используемой функции).

При проектировании VBA исходной целью проекта было создание сценарного языка для пользователей Excel, поэтому я был убежденным сторонником лагеря «слабой типизации». Очевидно, что такой подход проще понять непрофессиональному программисту, который плохо себе представляет, что такое переменная (не говоря уже о типе).

На моей стороне было сообщество Smalltak, которое в те дни выдвигало довольно туманные аргументы: «Вы все равно узнаете о проблеме, просто это произойдет на несколько секунд позже...» И это часто является правдой, но не всегда.

В конечном счете я одержал победу во внутренних дебатах в Microsoft; в VBA и COM был добавлен универсальный тип данных «Variant» — структура, способная содержать значение любого типа. Более того, позднее вышел VBScript, в котором поддерживался только универсальный тип, так что идея оказалась популярной.

И все же подсознательно я всегда знал, что сильная типизация — полезный механизм, который помогает компилятору выявлять множество разных ошибок. В C++ я всегда широко использовал систему типов. Например, если вы хотите быть твердо уверены в том, что рядовым работникам никогда, решительно никогда не должны выплачиваться премии, следует создать систему типов с классами

начальников и работников, в которой метод PayBonus() определен только для начальников. Если программа откомпилируется, можно не сомневаться: премии достанутся только достойным и благородным начальникам, а не жадным работникам.

Проблема в том, что создавать типы только для проведения дополнительных проверок на стадии компиляции немного неудобно. Типы позволяют выполнять только один вид проверок, а именно: «Можно ли применить эту операцию к этому объекту?» Нельзя проверить условие вида: «Действительно ли эта функция возвращает 2.12 для входных значений 1, 32 и 'aardvark'»?

Фактически задача превращается в головоломку для программиста, который должен придумать хитроумную схему типов, которая бы позволяла проверить некоторый мелкий аспект правильности программы.

Оказывается, для проверки правильности программ имеется другой, более мощный и прямолинейный инструмент: модульное тестирование. Поэтому идея Брюса Эккеля о сильном тестировании как замене для сильной типизации меня очень заинтриговала.

Прежде чем передавать слово Брюсу, я должен предупредить, что динамическая типизация заметно снижает быстродействие. Поскольку типы приходится анализировать и проверять во время выполнения, языки с динамической типизацией всегда работают медленнее языков со статической типизацией. Иногда это приемлемо, иногда — нет; все зависит от приложения. Обязательная динамическая типизация делает Python очень медленным языком. Я использую спам-фильтр, написанный на Python; и чтобы одно сообщение было помечено как спам, мне часто приходится ждать несколько секунд. А если помечается 10 или 20 сообщений, прямые потери от удобной «динамической типизации» составляют минуту или две. Если вы управляете работой комплекса веб-серверов, применение языка с динамической типизацией может означать, что для обслуживания того же числа клиентов потребуется в 5–10 раз больше серверов. Это может обойтись очень дорого.

Так что руководствуйтесь собственным здравым смыслом относительно того, какое быстродействие необходимо для вашего приложения. Но если модульные тесты обеспечивают хорошее покрытие кода, не нужно слишком комплексовать по поводу отказа от проверки типов на стадии компиляции. — *Ред.*

За последние годы наибольший интерес для меня представляла проблема производительности труда программиста. Ресурсы процессора стоят дешево, а ресурсы программиста — дорого, и я полагаю, что за первое не стоит расплачиваться вторым. Как добиться максимального эффекта в отношении решаемой задачи? Каждый появляющийся новый инструмент (особенно язык программирования) обеспечивает некий новый уровень абстракции, который может скрывать или не скрывать лишние детали. Тем не менее, я всегда остерегаюсь заманчивых предложений, особенно когда меня призывают не обращать внимания на выкрутасы, необходимые для достижения этой абстракции. Превосходным примером служит Perl — прямота этого языка скрывает бессмысленные подробности построения программы, но нечитаемый синтаксис (я знаю, обеспечивающий совместимость с такими инструментами UNIX, как awk, sed и grep) оказывается слишком высокой ценой. Последние несколько лет прояснили смысл этой «Фаустовой сделки» в контексте более традиционных языков программирования и их ориентации к статической проверке типов. Все началось с двухмесячного романа с Perl;

я получил желанную производительность (история закончилась из-за отвратительной работы со ссылками и классами в Perl; только позднее я увидел, что истинные проблемы кроются в синтаксисе). Проблемы выбора между статической и динамической типизацией не видны в Perl, потому что вам все равно не удастся построить достаточно большой проект, чтобы эти проблемы проявились, а в небольших программах все скрывает синтаксис.

После перехода на Python (распространяется бесплатно по адресу www.Python.org) — язык, который может использоваться для построения больших, сложных систем — я начал замечать, что при явно беспечном отношении к проверке типов программы Python обычно неплохо работают, не требуют особых усилий и лишены недостатков, которые можно было бы ожидать от языка без статической проверки типов (которая, как мы «знаем», является единственно верным способом решения проблем в программировании).

Я столкнулся с загадкой: если статическая проверка типов настолько важна, то почему людям удается создавать большие, сложные программы Python (с гораздо меньшими затратами времени и усилий по сравнению со статическими решениями) без катастрофических последствий, которые казались неизбежными?

Это обстоятельство поколебало мою абсолютную веру в статическую проверку типов (обретенную при переходе с языка C до выхода стандарта ANSI на C++, сопровождавшемся радикальными улучшениями) до такой степени, что когда я в следующий раз изучал проблемы проверяемых исключений в Java[1], я спросил «почему?». Последовала большая дискуссия[2]; мне объяснили, что если я буду выступать за непроверяемые исключения, падут города, а цивилизация в том виде, в котором мы ее знаем, перестанет существовать. В книге «Философия Java»[3] я пошел еще дальше и продемонстрировал использование RuntimeException как класса-обертки для «отключения» проверяемых исключений. Сейчас это уже воспринимается вполне нормально (помню, что Мартин Фаулер выдвинул ту же идею примерно в то же время), но время от времени я получаю по электронной почте сообщения о том, что я попираю все Правильное и Истинное, а заодно и нарушаю Патриотический Акт США (парни из ФБР, привет! Добро пожаловать в мой блог!).

Но даже если решить, что от проверяемых исключений больше хлопот, нежели реальной пользы (от проверки, а не исключений — на мой взгляд, единый и последовательный механизм сообщения об ошибках абсолютно необходим), это не даст ответа на вопрос: «Почему Python так хорошо работает, когда по всем правилам он должен приводить к массовым сбоям?» Python и другие языки с динамической типизацией весьма лениво относятся к проверке типов. Вместо того, чтобы устанавливать максимально жесткие ограничения на тип объектов на самой ранней стадии (как в Java), языки вроде Ruby, Smalltalk и Python накладывают минимальные ограничения на типы и проверяют их только при необходимости.

---

[1] Проверяемые исключения — возможность языка. Компилятор на стадии компиляции убеждается в том, что каждая функция содержит код, который либо обрабатывает любое возможное исключение, либо отказывается от его обработки, чтобы ответственность можно было передать другой стороне. — *Примеч. ред.*

[2] См. http://www.mindview.net/Etc/Discussions/CheckedExceptions.

[3] Эккель Б. Философия Java. 3-изд. — СПб.: Питер. 2003.

Так возникает идея латентной, или структурной типизации, часто называемой «утиной типизацией» (от поговорки: «Если оно ходит как утка и крякает, как утка, будем относиться к нему как к утке»). Это означает, что любому объекту можно отправить любое сообщение, а язык интересует лишь то, сможет ли объект принять это сообщение. В отличие от Java, язык не требует, чтобы объект относился к определенному типу. Например, программа с говорящими животными на Java выглядит так:

```
// Говорящие животные на языке Java:
interface Pet {
void speak();
}
class Cat implements Pet {
public void speak() { System.out.println("meow!"); }
}
class Dog implements Pet {
public void speak() { System.out.println("woof!"); }
}
public class PetSpeak {
static void command(Pet p) { p.speak(); }
public static void main(String[] args) {
Pet[] pets = { new Cat(), new Dog() };
for(int i = 0; i < pets.length; i++)
command(pets[i]);
}
}
```

Обратите внимание: метод `command()` должен знать точный тип получаемого аргумента (`Pet`), и ничего другого не примет. Таким образом, я должен создать иерархию `Pet` и объявить производные классы `Dog` и `Cat`, которые затем подвергаются повышающему преобразованию при вызове обобщенного метода `command()`.

В течение долгого времени я считал, что повышающее преобразование является неотъемлемой частью объектно-ориентированного программирования, и меня раздражали вопросы от невежественных сторонников Smalltalk и иже с ними. Но когда я начал работать с Python, обнаружилась одна любопытная подробность. Приведенный ранее код можно напрямую перевести на Python:

```
# Говорящие животные на языке Python:
class Pet:
def speak(self): pass
class Cat(Pet):
def speak(self):
print "meow!"
class Dog(Pet):
def speak(self):
print "woof!"
def command(pet):
pet.speak()
pets = [ Cat(), Dog() ]
for pet in pets:
command(pet)
```

Если вы никогда не видели кода Python, вы заметите, что он наделяет новым смыслом понятие компактности кода, но в очень хорошем смысле. Вы думаете, что код C/C++ компактен? Выбросьте фигурные скобки — отступы уже имеют

интуитивный смысл для человеческого ума, поэтому их можно использовать для обозначения области видимости. Типы аргументов и возвращаемых значений? Пускай язык сам разберется! При создании класса базовые классы заключаются в круглые скобки. Ключевое слово def означает, что мы создаем определение функции или метода. С другой стороны, Python требует явно указывать аргумент this (здесь он называется self) при определении методов.

Ключевое слово pass означает: «А это мы определим позднее», так что его можно считать разновидностью ключевого слова abstract.

Объявление command(pet) всего лишь означает, что передается некоторый объект с именем pet, но не несет никакой информации о типе этого объекта. Дело в том, что тип объекта неважен, при условии, что для объекта можно вызвать speak() или другую функцию или метод, которые требуется вызвать. Это пример латентной типизации, которая вскоре будет рассмотрена более подробно.

Кроме того, command(pet) представляет собой обычную функцию, что вполне допустимо в Python. Другими словами, Python не настаивает на том, чтобы все функции принадлежали объектам, поскольку иногда нужна именно функция, а не метод.

В Python списки и ассоциативные массивы (они же отображения или словари) играют настолько важную роль, что они включены в основной синтаксис языка, поэтому для работы с ними не нужны никакие специальные библиотеки. Пример нам уже встречался:

```
pets = [ Cat(), Dog() ]
```

Команда создает два новых объекта типов Cat и Dog. При создании объектов вызываются конструкторы, но ключевое слово new не нужно (если вернуться к Java, вы поймете, что и здесь без new можно обойтись — это всего лишь пережиток, унаследованный от C++).

Перебор последовательности тоже играет настолько важную роль, что в Python для него предусмотрена специальная операция; команда

```
for pet in pets:
```

последовательно заносит каждый элемент списка в переменную pet. На мой взгляд, такое решение гораздо четче и прямолинейнее решения, используемого в Java, даже по сравнению с синтаксисом J2SE5 «foreach».

Программа выводит тот же результат, что и Java-версия. Становится понятно, почему Python часто называют «исполняемым псевдокодом». Он не только достаточно прост для псевдокода, но и обладает тем замечательным свойством, что он действительно исполняется. А это означает, что вы можете быстро опробовать новую идею в Python, а когда она заработает — переписать ее на Java/C++/C# или другом языке по вашему выбору. Хотя если вы понимаете, что проблема уже решена на Python, зачем ее переписывать? (По крайней мере, я обычно думаю именно так). Я привык на своих семинарах давать подсказки к упражнениям на Python, потому что при этом я не выдаю всю картину, но слушатель видит форму поиска решения и может двигаться вперед. Кроме того, я могу проверить правильность псевдокода, запустив его на выполнение. Но самое интересное в другом: поскольку метод command(pet) не обращает внимания на тип полученного

объекта, повышающее преобразование не нужно. Следовательно, программу на Python можно переписать без использования базовых классов:

```
# Говорящие животные на языке Python, но без базовых классов:
class Cat:
def speak(self):
print "meow!"
class Dog:
def speak(self):
print "woof!"
class Bob:
def bow(self):
print "thank you, thank you!"
def speak(self):
print "hello, welcome to the neighborhood!"
def drive(self):
print "beep, beep!"
def command(pet):
pet.speak()
pets = [ Cat(), Dog(), Bob() ]
for pet in pets:
command(pet)
```

Поскольку command(pet) интересует лишь возможность отправки сообщения speak() своему аргументу, я убрал базовый класс Pet и даже добавил совершенно посторонний класс Bob. Так как этот класс содержит метод speak(), он тоже работает с функцией command(pet).

В этот момент язык с сильной типизацией уже бушевал бы от ярости, настаивая, что подобная неразборчивость совершенно недопустима. Само собой, в какой-то момент «неправильный» тип может быть использован при вызове command() или иным способом просочится в систему. Преимущества более простого и четкого выражения этих концепций попросту перевешивают риск — даже только это преимущество обеспечивает производительность, в 5–10 раз превышающую производительность программирования на Java или C++. Что произойдет, если такая проблема возникнет в программе Python — объект каким-то образом окажется там, где его быть не должно? В Python вся информация об ошибках выдается в виде исключений, как в Java и C#, и как это должно быть в C++. Таким образом, проблема будет обнаружена, но практически всегда это происходит во время выполнения. «Ага! — скажете вы. — В этом все дело: отсутствие необходимой проверки на стадии компиляции не позволяет гарантировать правильность программы».

Во время работы над книгой «Философия C++» (Thinking in C++) я реализовал очень примитивную форму тестирования: я написал программу, которая автоматически извлекала весь программный код из книги (начало и конец каждого листинга определялись по специальным маркерам, включаемым в листинг), а затем построил make-файлы, которые компилировали весь код. Это позволило мне гарантировать, что весь код в книге успешно компилируется, и я мог с полным правом сказать: «Если этот фрагмент взят из книги, значит, он правилен». Я не обращал внимания на назойливый голос, говоривший: «Компилируется — еще не значит правильно выполняется», потому что сама автоматизация проверки кода

была достаточно серьезным делом (как известно каждому читателю книг по программированию, многие авторы все еще уделяют недостаточно внимания проверке правильности кода). Естественно, некоторые примеры работали неверно, и когда за прошедшие годы у меня накопилось достаточно сообщений об ошибках, я начал понимать, что игнорировать проблему тестирования невозможно. Я начал так серьезно к ней относиться, что в третьем издании «Философии Java» написал:

«Если программа не протестирована, она не работает».

Я имел в виду, что если программа компилируется в языке со статической типизацией, это говорит всего лишь о том, что она прошла некоторые тесты. Это означает, что ее синтаксис заведомо правилен (Python тоже проверяет синтаксис на стадии компиляции — просто в этом языке меньше синтаксических ограничений). Но то обстоятельство, что код прошел проверку компилятора, еще не дает гарантий его правильности. Если программа запускается, это также не гарантирует ее правильности.

Независимо от того, какая типизация используется в вашем языке программирования — статическая или динамическая, единственной гарантией правильности является *прохождение всех тестов, определяющих правильность программы*. И некоторые из этих тестов вам придется написать самостоятельно. Конечно, это модульные тесты, приемочные тесты и т. д. В «Философии Java» я наполнил книгу своего рода модульными тестами, и потраченные усилия многократно окупились. Стоит «заразиться тестированием», и вы уже не избавитесь от этой привычки.

Происходящее очень напоминает переход с C на C++. Вдруг компилятор начинает выполнять многие проверки за вас, а программа быстрее переходит в работоспособное состояние. Однако возможности проверки синтаксиса ограничены. Компилятор не знает, как должна работать программа, поэтому его необходимо «расширить» определением модульных тестов (независимо от используемого языка). После этого вы сможете достаточно быстро вносить радикальные изменения (такие, как переработка кода или изменение архитектуры) — ведь при возникновении любых проблем тесты перестанут проходить, подобно тому, как при проблемах с синтаксисом не проходит компиляция. Без полного набора модульных тестов (по меньшей мере) гарантировать правильность программы невозможно. Конечно, утверждения о том, что статическая проверка типов в C++, Java или C# спасает от написания неверных программ, являются иллюзией (наверняка вы знаете это по личному опыту). Фактически нам нужно

Сильное тестирование вместо сильной типизации

Полагаю, именно благодаря этому аспекту работает Python. Все тесты C++ выполняются на стадии компиляции (за несколькими второстепенными исключениями). Одни тесты Java применяются на стадии компиляции (проверка синтаксиса), другие — на стадии выполнения (например, проверка границ массивов). Большинство тестов Python применяется на стадии выполнения вместо стадии компиляции, но важно здесь именно то, что они *применяются* (а не то, когда это происходит). А возможность создать рабочий прототип программы на Python

гораздо быстрее, чем при написании эквивалентной программы C++/Java/C, позволяет быстрее приступить к реальному тестированию: модульным тестам, проверкам гипотез, тестированию альтернативных подходов и т. д. А если для программы Python написаны нормальные модульные тесты, она по надежности не уступит программам C++, Java и C# с нормальными модульными тестами (хотя тесты на Python пишутся быстрее).

Роберт Мартин — один из ветеранов сообщества C++. Он является автором многих книг и статей, занимался консультациями, преподаванием и т. д. Активный сторонник статической проверки типов. Во всяком случае, я так считал, пока не ознакомился с записью в его блоге (http://www.artima.com/weblogs/viewpost.jsp?thread=4639 – *Ред.*). Роберт пришел более или менее к тем же выводам, что и я, но по иному пути: сначала он «заразился тестированием», потом понял, что компилятор был всего лишь одной (неполной) формой тестирования, потом — что языки с динамической типизацией могут быть гораздо более производительными, а при качественном тестировании написанные на них программы не уступают по надежности программам, написанным на языках со статической типизацией. Конечно, Мартин тоже получил обычные комментарии вида «Да как тебе в голову такое пришло?» Именно с этого вопроса когда-то началась моя борьба с самими концепциями статической/динамической типизации. И конечно, оба мы вначале выступали на стороне статической проверки типов. Интересно, что для переоценки предыдущих убеждений должно произойти что-то глобальное — такое, как «болезнь тестирования» или изучение новой разновидности языка.

Пол Форд

Размышления о Processing

Я давно принадлежу к числу поклонников сайта Пола Форда, Ftrain.com. Когда издательство APress предложило мне собрать материалы для этой книги, я знал, что книга должна включать одну из его статей.

В этой статье есть нечто, абсолютно точно воспроизводящее сущность того, что я считаю «элегантностью» в технических статьях. Ясность и поэтичность; лаконичные, четкие размышления; эрудиция; потрясающая, умопомрачительная концовка... Кажется, я упаду в обморок; дайте стул.

А может быть, автор просто иронизирует, но это тоже хорошо. — *Ред.*

Полуночные размышления о маленьких компьютерных языках, о Web как форме и о моем собственном невежестве.

Я некоторое время возился с Processing[1] — маленьким компьютерным языком, построенным на базе Java. Processing позволяет быстро создавать интересные (хочется надеяться) изображения и анимации, вроде представленного на прошлой неделе Square/Sphere/Static[2] или вчерашнего Red Rotator[3]. Пока я только осваивал Processing, но эта система интересна, проста в изучении, а в ее поставку входит простенькая, но умная IDE, позволяющая откомпилировать аплет щелчком на кнопке Play.

Программные конструкции Processing логически целостны и хорошо продуманы — в сущности, это упрощенный вариант Java, хотя слово «упрощенный» здесь неуместно. Правильнее было бы сказать «более элегантный», потому что авторы Processing определили для себя целевую аудиторию — дизайнеры компьютерной графики — и создали на базе пышной среды Java нечто такое, что дизайнер может быстро освоить и немедленно использовать. В сущности, они стесали трон Java, вырезанный из цельного дуба, и превратили его в изящный стул с алюминиевым каркасом и упругой спинкой.

---

[1] См. http://processing.org/.

[2] См. http://www.ftrain.com/SquareSphereStatic.html#ProcessingProcessing.

[3] См. http://www.ftrain.com/RotatingSquares.html#ProcessingProcessing.

Почему я завел об этом речь? Дело в том, что у меня есть пристрастие к языкам типа Processing, о котором я предпочитаю не говорить в вежливых или легко утомляющихся компаниях[1] — к языкам, компилирующимся не в исполняемый код, а в эстетичные объекты, будь то графические изображения, песни, демонстрационные ролики или веб-сайты. К специализированным языкам такого рода относится CSound, компилируемый в звуковые файлы; POV-Ray, компилируемый в трехмерные изображения; TeX, компилируемый в макеты, готовые для типографии; SVG (Scalable Vector Graphics) — схема XML для создания векторной графики, и т. д.

Также существуют другие, более общие языки, направленные на удовлетворение потребности особой категории программистов: язык ActionScript лежит в основе Macromedia Flash и получил повсеместное распространение в Web; язык Грэхема Нельсона (Graham Nelson) Inform с большой библиотекой усовершенствований, разработанной сообществом пользователей, компилируется в интерактивные текстовые приключения. В дальнем конце спектра находятся языки общего назначения типа C, Java, Perl и Python — языки, позволяющие делать все, на что способен компьютер. Processing существует где-то посередине. С одной стороны, он является языком программирования общего назначения (особенно если учесть, что он позволяет вызывать любые функции Java), но при этом он ограничивается очень небольшим набором примитивов — точки, сферы, прямоугольники и т. д. — и несложной моделью трехмерного пространства. Код Processing компилируется в весьма специфический объект: интерактивный графический элемент. Processing сильно напоминает Inform своей ориентацией на конкретную область применения: скажем, на Inform не удастся написать текстовый редактор — и Processing для этой цели тоже не подойдет. Но если вы хотите создать текстовое приключение, Inform становится хорошим кандидатом, гораздо лучше простого C. Аналогично, если вы хотите создать графический объект 200 × 200 с поддержкой щелчков мышью, Processing тоже отлично подойдет для этой цели.

Упомянутые мной языки вознаграждают усилия, потраченные на изучение, потому что они представляют точку соприкосновения между эстетикой и компьютерными технологиями. Например, в CSound имеется файл партитуры и файл инструментов оркестра; последний содержит набор инструментов, звуки которых синтезируются на основе гармонических осцилляторов, звуковых сэмплов и всевозможных временны*х* конструкций: сигналов, линий и волн. Файл партитуры представляет собой коллекцию мелодий, ритмов и переменных, передаваемых инструментам. На примере такого языка можно многому научиться; он представляет целенаправленную попытку создания творческой грамматики, ограниченной тремя факторами:

- возможностями компьютера по эффективной обработке некоторых типов данных;
- склонностями разработчиков языка и их понимания выбранной предметной области;
- готовности рядового программиста работать в рамках двух перечисленных ограничений.

---

[1] Теперь вы знаете, на какую аудиторию я ориентируюсь.

Я не хочу сказать, что все должны изучать эти языки, но если вы, как и я, хотите лучше понять, на что способны компьютеры в области мультимедиа, а также понять культурные факторы построения инструментария для создания мультимедийного содержания, вы обнаружите, что эти языки являются чрезвычайно интересным объектом для изучения.

CSound был первым языком программирования, который я изучил в 1996 году. Электронная документация была крайне неполной, и мне пришлось пойти в библиотеку, чтобы изучить теорию осцилляторов и понять различия между аддитивным, вычитательным и гранулярным синтезом. В конечном итоге я собрал самодельный осциллограф из старого телевизора для визуального представления энергетического спектра звуков; он помогал мне разобраться, почему синусоидальный сигнал так сильно отличается на слух от пилообразного. Один из откомпилированных мной файлов CSound строился более 20 часов, потому что в нем были задействованы десятки тысяч интерактивных инструментов, манипулировавших друг другом, с реверберацией по всему звуковому спектру, воспринимаемому человеческим ухом. Результат звучал ужасно; музыканта из меня не выйдет. Но мне было интересно заглянуть внутрь звука при помощи этого маленького языка.

Глядя на Processing, я вижу, что многое из того, что я узнал из CSound, перенеслось в визуальную область (Processing поддерживает работу со звуком, но лишь на минимальном уровне). Осциллятор в CSound является аналогом цикла for в Processing; в коде, написанном мной вчера, квадрат вращается вокруг фиксированной точки, при этом в каждом кадре он смещается на несколько пикселов вперед. В CSound я мог бы определить серию осцилляторов, модулирующих друг друга; изменение одного осциллятора могло бы привнести тремоло в шумные аккорды другого осциллятора. В Processing циклически изменяющиеся значения можно суммировать (со вставкой данных с мыши или другого источника), чтобы вместо введения вибрации в звук синтезатора красные квадраты начинали перемещаться по кругу. Но идея остается прежней: значения изменяются со временем, увеличиваются и уменьшаются, и эти регулярные изменения используются для достижения интересного или красивого эффекта, будь то перемещение объекта или смена частоты. Этот способ работает, потому что людям нравятся красивые движущиеся картинки и звуки, изменяющие частоту и амплитуду с регулярными интервалами.

Processing вернул меня к 14-летнему возрасту, когда я играл с режимом анимации Deluxe Paint на Amiga и учился вращать текст вдоль осей *X*, *Y* и *Z*, и в течение многих часов учился основам перспективы и геометрии (как спонтанно, так и потому, что это было интересно). Я долгие годы искал нечто похожее в отношении визуальной гибкости, и наконец-то нашел Processing.

После всего сказанного у меня остается вопрос: если существуют языки для определения инструментов и осцилляторов, линий и сплайнов, и даже языки для реализации концепций верстки вроде TeX, почему не существует общепринятой системы веб-публикаций?

Я знаю, что в мире существуют тысячи систем управления содержанием, от Midgard до Movable Type, причем каждая система имеет свое конкретное представление о содержании. В этих системах используются базы данных; они сортируют

информацию по дате и времени, по автору и категории; в них реализованы теги XML, схемы и DTD. Но единой инфраструктуры для них не существует. Я уже говорил об этом в Web Pidgin[1], но могу пояснить дополнительно: при построении сайта FTrain использовалась пользовательская схема XML, XSLT (под этим термином в действительности объединяются два языка — язык преобразований XSLT [Extensible Stylesheet Language Transformations] и язык доступа к дереву документа XPath); Makefile; технология XHTML1.1, определяющая структуру страницы; таблица CSS, определяющая внешнее представление XHTML1.1, а также код JavaScript, определяющий некоторые интерактивные возможности страницы. В конечном итоге все это экспортируется в RSS0.91, RSS1.0, RSS2.0 и Atom, а полная копия сайта будет выведена в формате RDF (Resource Description Framework). Сайт содержит аплеты Java, звуковые файлы в формате RealAudio и MP3, изображения JPEG, GIF, PNG, текстовые файлы, сценарии Python, сценарии Perl, страницы PHP и поисковый механизм.

И это один сайт для одного человека. Выходит слишком много.

Система TeX необычайно гибка в определении структуры книги и ее содержимого, и она использовалась при подготовке многих тысяч публикаций. Но, в конечном счете, она состоит из двух маленьких специализированных языков: TeX для определения макета и MetaFont для определения глифов, с различными подъязыками и библиотеками для их возможного расширения. Такие пакеты, как Adobe's InDesign, Photoshop и Illustrator, также пытаются решить аналогичную проблему: они предоставляют однородную среду, в данном случае — среду, в которой вы вместо программирования указываете и щелкаете кнопкой мыши. Проблема решается в одном месте, в одной среде, с единым набором инструментов.

Веб-сайты не сложнее в создании, чем книги (а во многих отношениях гораздо проще), но процесс создания книги закодирован и четко определен; в нем существуют свои нормы, четкое разделение труда и понимание того, что должно следовать дальше в любой момент времени. Прочитайте несколько руководств по типографскому делу, посетите типографию, взгляните на Гейдельбергский печатный станок и поговорите с редактором крупного издательства. Но если уже обладаете опытом публикаций в Web, процесс покажется вам до отвращения медленным и негибким — например, спрос на последнюю повесть о Гарри Поттере отодвинул на второй план десятки других книг, чтобы сделать возможным поставку многомиллионного тиража первого издания книги Роулинг. В Интернете в таких ситуациях можно просто установить несколько новых серверов, купить более мощный канал и удовлетворить спрос.

Однако вам придется определить, на каких пользователей ориентироваться в первую очередь, продумать всевозможные процессы переключения с сохранением высокой доступности сайта; это можно сделать, но непросто. Так что все верно, в издательском деле гибкости меньше, но меньше и рыданий в ужасе. Потому что процесс веб-разработки действительно внушает ужас. В нем не существует момента, когда вы могли бы с полной уверенностью сказать: «Все готово».

---

[1] См. http://www.ftrain.com/WebPidgin.html#ProcessingProcessing.

Я до сих пор постоянно получаю жалобы от пользователей Macintosh Internet Explorer 5.2. Я опробовал с десяток мер, которые должны были обеспечить работу сайта в выбранном ими браузере, и все безуспешно[1]. Некоторые проблемы попросту невозможно решить независимо от затрат ресурсов, времени и исследований — а ведь всего этого вечно не хватает, особенно если ты должен заставить сайт работать к 9 вечера в воскресенье.

Мне кажется, проблема отчасти обусловлена тем, что веб-разработчики все еще страдают синдромом гордыни новой экономики, полагая, что они обладают особым даром, глубинной мудростью, выходящей за рамки всего ранее существовавшего, и что их существование оправдывает пророчества Маклуэна. Исключение составляет масса умнейших людей, которые не обращали внимания на CSS и работали на некоммерческие организации. И многие из них, в отличие от многих из нас, продолжают заниматься тем, что им нравится. Невольно возникает вопрос: так ли уж замечательна Web, если многие из ее стойких поклонников не могут заработать на достойную жизнь, трудясь над ее совершенствованием? Думаю, настало время сделать шаг назад и спросить: «Действительно ли игра стоит свеч?» Конечно, вы можете догадаться, каким будет мой ответ[2], но я думаю, этот вопрос все же стоит задать.

Глядя на Processing, я ловлю себя на мысли: мне бы хотелось, чтобы Web работала так же. Нет, я вовсе не стремлюсь превратить Web в набор маленьких графических объектов, на которых можно щелкать мышью; мне хотелось бы, чтобы люди сделали шаг назад, взглянули на все сделанное и придали элегантность Web как конструкции, определили набор базовых проблем, которые захотели бы решать веб-разработчики, и создали бы маленький язык, основанный на минимально возможном наборе принципов, который помогал бы в решении этих проблем. На этой стадии моей жизни как веб-разработчика мне не нужны учебники по настройке CSS, чтобы они нормально выглядели в IE 5.2 для Macintosh (после десятков потраченных часов я намерен отказаться от дальнейших попыток); скорее, мне нужен ответ на вопрос «Что такое ссылка?» Мне не нужно, чтобы кто-то попытался упростить мою работу за счет создания очередного Dreamweaver или FrontPage; я хочу, чтобы она стала элегантной, как элегантен язык Scheme. Я хочу знать:

1. Что такое веб-страница? Где она начинается и кончается? Насколько полезна эта концепция, или веб-страницу лучше рассматривать как одно из представлений гораздо большей базы данных взаимосвязанных документов?
2. Насколько хорошо браузер подходит для работы в Web? Он удобен для просмотра страниц HTML, но я бы более охотно использовал базу данных/электронную таблицу для поиска в Web и моих локальных файлах, которая бы запускала браузер тогда, когда это потребуется. Другими словами, что-то вроде Google, но внутри Excel. Огромную часть веб-содержания составляют метаданные — поисковая информация, оглавления и т. д. И если сайт

[1] По правде говоря, этот сайт — самое дурацкое увлечение, которое я только мог выбрать.

[2] Я написал единственную в мире 200-мегабайтовую персональную рекламу в форме веб-сайта.

может быть представлен одним крошечным значком на панели навигации URI, не было бы удобнее иметь единую систему навигации, расположенную у верхнего края сайта?

3. Чем *em* лучше *i*? Когда я публикую информацию 1901 года, и она выделена курсивом, это именно курсив, а не выделение. У типографского дела имеется своя семантика — тонкая, изменяющаяся и сформированная под воздействием исторических факторов. Текущее состояние Web более или менее полностью игнорирует это обстоятельство и постоянно пытается кодировать типографские стандарты и идеи в виде древовидных структур данных, вроде тега <*q*>.

Почему некоторые семантические конструкции пользуются большими привилегиями, чем другие? Почему теги *blockquote*, *em*, *strong* и *q* важнее несуществующих тегов *event*, *note*, *footnote* или *fact*? Потому что HTML пытается наследовать подразумеваемую типографскую семантику, вот почему! А эта семантика гораздо тоньше и сложнее, чем кажется многим людям (за пределами сообщества TEI и подобных им). Но приверженность этой семантике означает, что Web — ориентированная на типографские стандарты, семантически незрелая система... безумие, сущее безумие.

Как организовать полноценное многократное использование содержания? Я говорю не о преобразовании DocBook XML в книгу или набор веб-страниц, а о вставке отдельных предложений и фраз в временные диаграммы, автоматическом извлечении диалогов из коротких историй, и других вещах такого рода.

Если ссылки наделяются семантикой, чтобы вместо концепции «Ссылка на эту страницу» реализовывалась концепция «Эта страница является расширением этой страницы» или «Автором этой страницы является ресурс X», что нам это дает?

Вообще зачем нужен браузер? Недавно я нашел огромную базу данных отсканированных журналов с 1800 по 1900 год. Ее содержимое было собрано в большие оглавления, которые мне совершенно не хотелось просматривать. Поэтому я выполнил автоматическую обработку базы данных и построил собственное оглавление, которое я перенес в базу данных (и которое мой друг Кендалл Кларк преобразовал в формат RDF, чтобы его можно было использовать в приложениях семантической сети; я постараюсь вскоре опубликовать его).

Последние три вопроса для меня не совсем корректны, потому что над их решением я усердно работал последние два месяца. Мне удалось разработать ряд решений, которые я опишу в одном из будущих очерков. Но я сомневаюсь, что мои решения очень хороши; они всего лишь необходимы для того, чтобы я мог делать то, что мне нужно. Кому-то будет интересно узнать, что я собираюсь распространять весь сайт (объем которого вскоре превысит 1 000 000 слов) в обычном формате RDF с присоединенной базой данных цитат, событий и т. д., отображенных из содержания. Если кто-то захочет просмотреть Ftrain (или похожий сайт) в другом формате, он может просто написать тот интерфейс, который наилучшим образом подойдет для него. Я планирую переместить управление ресурсами в электронную таблицу. А еще я хочу купить себе хорошие носки и колокольчик на велосипед.

Итак, я надолго задумался о возможности создания CSound или Processing для Web. Этот инструмент должен понимать концепцию ссылок и очень специфические потребности дизайнеров, информационных архитекторов и читателей/пользователей сайта; он не должен попадать под влияние конкурирующих традиций из области проектирования интерфейса, издательского дела, журналистики и типографии. Нам нужно нечто такое, что позволит нам воспринимать Web как единое пространство, а не как набор из интерфейсов представления (CSS), языков преобразований (XSLT), механизмов, ориентированных на структуры данных (DOM, XPath), спецификаторов интерфейсов (JavaScript) и решений, основанных на разметке (XHTML).

Один из возможных сценариев виден на примере REST (Representational State Transfer). Архитектура REST для Web представляет собой попытку «приведения к элегантности» того, что до появления формального описания было неудобным и предназначалось для конкретных случаев. REST представляет собой механизм, который описывает смысл адресов URI (типа http://ftrain.com), возможность их использования для генерирования запросов по сети, а также представления Web в виде совокупности не страниц, а программ, подключение к которым осуществляется через URI. Сравните REST, систему простую и реально работающую, с вебслужбами, которые добавляют новый уровень сложности в существующую структуру Web, существуют параллельно с подсистемой Web, ориентированной на содержание, и происходящие от представлений о распределенных объектах и сетевых вычислениях, появившихся раньше Web.

Оба механизма пытаются сделать примерно одно и то же. Но на мой взгляд, причиной успеха REST и относительной неудачи веб-служб является то, что REST в полной мере строится на базе Web. Разработчики взяли то, что работало, и сделали его более элегантным: простым для формального понимания, простым для обучения. Элегантность не является снобистской глупостью; это способ описания идей и жизнеспособных решений, выходящих за рамки текущих потребностей, способных внести новый вклад в отрасль и послужить основой для дальнейшего развития, в отличие от простого применения к существующей проблеме и забвения. Технология REST обладает этими качествами: она улучшила то, что существовало до нее.

Аналогичная проблема возникает и с Семантической Паутиной. Инфраструктура семантической сети направлена на решение проблем, важных для сообщества исследователей в области искусственного интеллекта, но не столь существенных для всех остальных. Менее мощные, но более привычные альтернативы типа SHOE (Simple HTML Ontology Extensions), позволяющие внедрять логические данные в HTML, были отложены в сторону для создания технологии, способной решать гораздо более широкий круг проблем: комбинации RDF/RDFS/OWL. Но когда сообщество, внесшее значительный вклад в формирование набора идей и привычек (в данном случае сообщество, занимающееся анализом представления знаний), занимается определением стандарта, возникает серьезная проблема. Эти люди занимаются тем, что не представляет интереса для большинства других людей; они строят обобщенные системы, воплощающие десятилетия исследований, и предвидят сотни сложных проблем, о существовании которых никому не известно.

Ничего особо ужасного в этом нет, но при таком подходе начинаются странные диалоги между разработчиками стандартов и внешним миром. В случае с семантической сетью диалог звучит примерно так:

ВНЕШНИЙ МИР. Мне хотелось бы сделать свой сайт «умнее», чтобы ссылки стали более содержательными.

СООБЩЕСТВО СЕМАНТИЧЕСКОЙ СЕТИ (ССС). Ну конечно! Тебе нужна обобщенная инфраструктура для онтологической разработки.

ВНЕШНИЙ МИР. Ладно. А это упростит создание ссылок?

ССС. Даже лучше, это приведет к образованию гигантского Всемирного Мозга, который будет планировать твои визиты к зубному врачу! Только прочитай эту статью в Scientific American.

ВНЕШНИЙ МИР. Это потребует больших усилий?

ССС. Нет. Но даже если и потребует, мы скажем, что ты слишком глуп и сам не понимаешь, зачем тебе это нужно.

ВНЕШНИЙ МИР. Угу. Наверное. Правда, я все равно не понимаю, зачем мне это нужно.

ССС. Потому что ты слишком глуп. Все нормально, мы действуем ради твоего же блага.

ВНЕШНИЙ МИР. Не хочу казаться назойливым, но все-таки как насчет ссылок?

ССС. Ну если ты настаиваешь и не можешь подождать, у тебя всегда есть XLink.

ВНЕШНИЙ МИР. Ага. Смотрится неплохо... правда, немного смущает отсутствие общедоступной реализации. И еще я не понял, что же все-таки должна делать эта технология.

ССС. Потому что ты глуп и ленив.

ВНЕШНИЙ МИР. Понятно. А может, мне подать заявку на грант на разработку моего маленького сайта FTrain.com? Мне много не нужно.

ССС. Все гранты будут у нас! А ты со своей степенью бакалавра из второразрядного частного колледжа можешь убираться подальше. И где твоя поставка RSS?

ВНЕШНИЙ МИР. Прошу прощения.

ССС. Тупица! И приведите мне еще аспирантов, я голоден!

Так или иначе, принципы работы семантической сети могут включать XML и транспорт HTTP, но они имеют очень мало общего с современной инфраструктурой Web, состоящей из страниц HTML. Мне потребовалось около 15 минут, чтобы полностью понять технологию SHOE, внедряемую в HTML. На осознание RDF мне потребовалось два года[1]. Я, конечно, не гений, но и не полный дебил, так

[1] Это напомнило мое правило: если вы не можете понять спецификацию новой технологии, не огорчайтесь — ее никто не поймет, и эта технология все равно не приживется. — *Примеч. ред.*

что два года — это слишком долго (кстати говоря, ключом к пониманию RDF может стать чтение учебника по языку Prolog; концепции те же самые, разобраться в них не так уж сложно — и после этого монолитная, зловещая глыба спецификации RDF начинает приобретать очертания).

В данном случае я не собираюсь ругать RDF — я использую эту технологию и мне придется полюбить ее, поверить в нее как в основополагающий механизм обмена данными и изобрести миллион идей для ее применения на FTrain. Но прежде чем я начну превозносить RDF до небес, мне также хотелось бы видеть определение этой технологии в контексте «повышения элегантности» существующей Web. На самом деле мне хотелось бы видеть все стандарты W3C и других организаций определяемыми в этой абстрактной, неконкретной манере, хотя этого никогда не случится: «Эта схема или стандарт делает нечто более элегантным и красивым, потому что...» Если бы этот простой критерий применялся на практике, стандарт XML Schema вообще не существовал бы, на протокол SOAP (Simple Object Access Protocol) все смотрели бы с глубоким подозрением, а REST и RelaxNG вошли бы в пантеон полезных стандартов.

Меня это все действительно волнует, потому что Web могла бы быть красивой. Сейчас она таковой не является. После бессчетных часов, проведенных за настройкой баз данных, подгонкой CSS и определения схем, после изучения RDF для дальнейшего заимствования идей и размышлений о том, что же такое ссылка, я могу с уверенностью сказать, что Web нельзя назвать красивой. В отношении зрелости технологий, которую можно измерить по способности технологии отражать реальные навыки и компетентность людей, которым она должна служить, Web является аналогом IBM PC Jr. Эквивалент абстрактных оконных интерфейсов еще не появился в этой среде. Когда я гляжу на книги по информационной архитектуре и пособия по построению веб-сайтов, они выглядят такими же смешными и архаичными, как руководство по матричным принтерам Epson в эпоху PostScript. Я не знаю, какие технологии придут на смену существующим, но готов поспорить, что они уйдут в небытие (как и все остальное, конечно).

Новые технологии обычно рождаются в малых группах напряженно мыслящих личностей. Возьмем оконные интерфейсы: для их создания потребовался исследовательский комплекс в стиле Xerox Palo Alto Research Center (PARC), несколько беспринципных бизнесменов в Apple и Microsoft, желающих нарушить чужие патентные права, и гениальные специалисты, которых можно загонять до полного изнеможения, чтобы сделать технологию более дешевой и доступной. Возьмите эти ингредиенты, добавьте несколько миллионов долларов — и пожалуйста, вы получаете компьютер, который изменит мир: Apple Lisa.

Но потом никто не захотел потратить 8 триллионов долларов на Lisa (и Apple II GS, и GEOS для Commodore 64, оснащавшие старые компьютеры поддержкой новых окон). Идея забуксовала. Mac продолжает существовать вместе со своим умственно отсталым братцем Windows и их общим другом X Windows, страдающим от синдрома раздвоения личности. Так что будет интересно посмотреть, откуда подует новый ветер в Web: кто поможет сконцентрировать идеи, и какой управленец с маниакально-депрессивным психозом сможет преобразовать результат в большую, рыночную концепцию, распространяющуюся со скоростью эпидемии.

А если попробовать рассматривать веб-сайт как *форму*? Возможно, это позволило бы нам создать язык вроде Processing, который бы помогал людям строить сайты; вместо новых стандартов мы имели бы библиотеки, подключаемые к среде разработки, как в TeX. Это стало бы концом системы из 30 с лишним стандартов, которыми мы сейчас вынуждены жонглировать, и которые основательно перекрываются.

Я говорю не о том, что будет работать или что произойдет, а лишь о том, что могло бы стать элегантным — о том, что позволило бы людям создавать красивые сайты. У меня есть несколько идей, воплощенных в Ftrain: я избавился от всего внутреннего структурирования сайта вроде секций, глав, авторов и описаний, и вместо этого выразил данные в RDF-подобном синтаксисе, подкрепленном (псевдо-) онтологией. Когда кто-нибудь захочет увидеть все истории на сайте, этот механизм позволяет просмотреть как документальные материалы, так и беллетристику; если кому-то потребуется просмотреть только беллетристику — что ж, и это тоже возможно. Это совершенно новое представление о сайте, и я не уверен в том, что я его полностью понимаю. Однако наличие внутренней онтологии структурирования содержания также дает мне массу новых идей по поводу навигации, чтения и т. д.

Я избавился от семантических ограничений уровня разметки вроде тегов *q* и *blockquote*, потому что они были плохи, и использую вместо них уникальные узлы с URI-адресацией. Таким образом, каждое событие, цитата, факт, измышление и т. д. является абсолютно уникальным. Я включил условный текст, чтобы цитата отображалась одним способом, допустим, в газетной статье, и совсем другим способом — в сборнике цитат. Скажем, статья может содержать строку: «Я уронил собаку, — сказал президент Буш. — О господи, я уронил собаку». Но на странице, посвященной Джорджу Бушу, эта цитата должна выглядеть иначе: *Я уронил собаку, о Господи, я уронил собаку. Джордж Буш.* Использовать один источник для создания обоих представлений не так просто, как кажется на первый взгляд. И пользователь должен иметь возможность взять один большой RDF-файл Ftrain, и еще один RDF-файл от кого-то, использующего тот же инструментарий построения сайтов, объединить их в одну большую базу контента с общей онтологией и просматривать так, как он считает нужным. Сейчас я усердно работаю над этим, в одиночку и в полном недоумении (а в это время мне приходят десятки сообщений с вопросом, куда делся мой RSS-файл; похоже, я неверно оцениваю приоритеты).

Зачем все это нужно? Потому что это интересно. Подобно тому, как CSound помог мне понять, что такое звук, построение моей собственной системы помогает лучше понять, что такое текст, что такое типографская разметка, и что они представляют собой в контексте Web. Это способ разрешения вековых противоречий между риторической традицией софистов и аристотелевской традицией. Текст, отображаемый на экране, представляет собой обычную прозу, которая должна течь гладко и представать во всем риторическом блеске. Но ссылки и данные, используемые для управления содержанием, представляют собой простые логические утверждения: Люди смертны. Сократ — человек. Следовательно, Сократ смертен. Пол Форд написал этот очерк. Следовательно, Пол Форд является писателем. Эта страница связана с той страницей.

Вы читаете текст, сконструированный на основе риторических принципов, находящийся под прямым и косвенным влиянием всей истории композиции от софистов до Дерриды. Но для перемещения по нему применяются простые логические утверждения, с блоками текста или изображениями, при щелчках или выделении которых на экране появляются другие файлы. Текст базируется на традициях риторики; ссылки базируются на традициях логики; и где-то здесь кроется нечто такое, в чем стоит разобраться поподробнее (причем соответствующие шаги предпринимали такие люди, как Ричард Лэнхем (Richard Lanham), разработчики системы PLINTH и другие).

Специалист по истории риторики Лэнхем указывает, что вся история западной педагогической школы может рассматриваться как колебания между двумя традициями: между традицией, рассматривающей риторику как средство получения (или критики) силы (язык как система взаимосвязанных знаков, находящихся вне морали и не имеющих основы в виде «истины»), и традицией поиска истины, поиска фундаментальных логических основ Вселенной, использования таких идей, как платоновы тела или булева логика, или таких инструментов, как экспертные системы и ускорители частиц. Вместо того, чтобы считать верной одну из этих двух традиций, пишет Лэнхем в своем «Электронном Слове», именно борьба этих двух традиций характеризует историю дискурса; эти колебания встроены в западную культуру, а при их обсуждении часто упоминается концепция «выражательства» (*sprezzatura*) — то есть заученной небрежности; работы, которая выглядит как выполняемая без усилий. Так появился мой сайт, позволяющий мне проработать эту проблему на практике: какая связь существует между повествованием и логикой? Что собой представляет *sprezzatura* для Web?

Понятия не имею. Мой способ разобраться в происходящем заключается в том, чтобы построить систему и работать над ней, потому что я слишком туп для разработки теорий. Я понятия не имею, что делаю, и большая часть этой работы делается с чувством полной безнадежности — когда, как сегодня вечером, я выдаю почти 4500 слов, представляющих мои лучшие мыслительные усилия, однако эти слова получаются твердыми, как прессованный творог, полны дыр, как швейцарский сыр, пахнут, как лимбургский сыр, обладают вкусом хорошего бри, мажутся, как плавленый сыр, покрыты слоем воска, как гауда, и многослойны, как сырный рулет.

Найдется много людей, гораздо умнее меня, которые будут вечно работать с языком, разбираться с семиотикой, разбивать язык на компоненты и определять, подобно Солу Крипке (Saul Kripke), что же такое «имя». Они используют уравнения и ищут истину. Я всего лишь ищу способ рассказать историю, который бы работал в обусловленных границах. Я хочу развлечь, заинтересовать, пробудить воображение своего читателя. Я слишком недалек, чтобы поверить в концепцию истины. А это означает, что я с открытым ртом глупо смотрю на сообщество искусственного интеллекта, на специалистов по лингвистике, экспертов по алгоритмам, разработчиков стандартов, специалистов по теории множеств, текстологов — но смотрю не с ревностью, а в депрессии, словно уличный гитарист, пытающийся со своими тремя аккордами сыграть увертюру из «Весны священной». Как бы я ни хотел все это вообразить, никакого сколько-нибудь полного понимания у меня

так и не получается. Я встречал людей, способных выдавать мысли длиной в несколько страниц — и я не принадлежу к их числу.

Впрочем, и у меня есть свои достоинства. Скажем, теперь в мире стало на 4500[1] слов больше. Это чего-то да стоит; хотя эти слова не блещут качеством, для кого-нибудь они могут оказаться полезным плохим примером. Возможно, несмотря на все недостатки изложения, они заинтересуют или развлекут кого-нибудь. А еще у меня есть система управления контентом, которая постепенно начинает работать, демонстрирует ограничения моей прозы, прокладывает путь для будущей работы и позволяет мне вытворять со словами некоторые вещи, которые ранее были невозможны — и делать это незаметно для большинства читателей, с ориентацией на идеи автора, а не на среду, в которой я работаю.

Вот что самое неприятное в новых средах: объем работы, связанной с обслуживанием самой среды. Отличным примером могут послужить блоги: достаточно большой процент блогов посвящен ведению блогов. Люди постигают, что можно делать с новыми формами, словно человек с микрофоном говорит: «Я говорю в микрофон, привет, я у микрофона, эай, микрофон. Микрофон. Эй. Это я. Я здесь. И говорю. Да, вот здесь, у микрофона. Говорю я тут. Пожалуйста, загляните в мой блог». Как вам скажут родители любого младенца, период нарциссического самосознания всегда является частью раннего роста, и пройдет немало лет, прежде чем мы избавимся от него в своих коллективных системах. И все же со временем люди поймут ценность высказываний, выходящих за рамки «Я что-то говорю», и с этого можно будет начинать. Говорят, среда передачи информации является сообщением... однако сообщение тоже является сообщением.

Что касается меня, то я думаю, что буду продолжать работу в этом направлении (пока не сломаюсь), страдая от того же синдрома созерцания собственного пупа, что и все остальные, придумывая способы передачи сигнала так, чтобы не увязнуть в сложностях передающего оборудования, и обходясь без бичевания собственного невежества более чем по несколько часов в день. Я всегда буду глуп, если посмотреть на широту людской мысли, но я постараюсь не комплексовать из-за этого, и вряд ли я когда-нибудь смогу остановиться, когда в мире осталось столько непознанных вещей. Даже глупец может сказать: знаете, это можно сделать красиво.

---

[1] Вообще-то ближе к 5600. — *Примеч. ред.*

Пол Грэхем

Великие хакеры[1]

Несомненно, программный проект не может завершиться успехом без участия суперзвезд программирования. Пол Грэхем называет этих суперзвезд «хакерами» и размышляет о том, что общее есть у всех хакеров. Когда эта статья была опубликована впервые, она вызвала массу споров — главным образом потому, что, по мнению Пола, программисты Java или программисты, пишущие код Windows, не могут быть хорошими хакерами. Я совершенно не согласен с этим утверждением; видимо, Пол не знает тех людей, которых знаю я. Известные мне великие хакеры также были самыми умными студентами; они посещали лучшие учебные заведения, где изучали Unix, и с тех пор придерживаются этой системы, потому что она им хорошо знакома. А многие люди работают на Visual Basic и Java, потому что они не являются великими хакерами, а эти языки позволяют быстро написать рабочую программу.

Но по-настоящему великие хакеры не привязываются к одному инструментарию и используют те средства, которые решают насущную задачу. Меня гораздо больше впечатляет человек, сделавший нечто замечательное в ужасной среде программирования, чем тот, кто не желает работать над нужной задачей, потому что решение не может быть выражено на языке Python.

Так или иначе, пусть это обстоятельство не отвлекает вас от очень важного очерка, написанного одним из лучших современных авторов, пишущих о разработке программного обеспечения. — *Ред.*

Несколько месяцев назад я закончил новую книгу. С тех я постоянно встречаю в рецензиях такие эпитеты, как «провокационная» и «спорная» (не говоря уже об «идиотской»). Я не собирался делать книгу спорной. Я всего лишь постарался сделать ее эффективной. Мне не хотелось попусту тратить чужое время, рассказывая людям уже известные истины — эффективнее показывать различия. Но похоже, такие книги всегда будоражат людей.

---

[1] Очерк написан по материалам доклада на конференции Oscon 2004.

Эдисоны

Относительно того, какая идея была самой спорной, никаких споров не было: я предположил, что различия в богатстве — не такая большая проблема, как мы думаем. Заметьте, в книге я не говорил, что различия в богатстве сами по себе хороши. Я всего лишь сказал, что в некоторых ситуациях они могут быть хорошим признаком. Например, головная боль сама по себе плоха, но иногда она оказывается хорошим признаком — скажем, если вы приходите в сознание после удара по голове.

Различия в богатстве могут свидетельствовать о различиях в производительности труда. А это почти всегда хорошо: если общество не различается в производительности труда, вероятно, это не потому, что оно состоит из одних Томасов Эдисонов. Скорее всего, в нем нет ни одного Томаса Эдисона.

В сообществах с низким уровнем технологического развития различия в производительности труда малозаметны. Если племя кочевников собирает хворост для костра, насколько лучший собиратель хвороста работает эффективнее худшего? Вдвое? Но если вручить человеку такой сложный инструмент, как компьютер, различия в производительности его труда могут быть огромными.

Эта идея не нова. Фред Брукс писал о ней в 1974 году[1], а исследования, на которые он ссылается, были опубликованы в 1968 году. Но мне кажется, он недооценил различия между программистами. Он писал о производительности в объеме кода: лучшие программисты могут решить конкретную проблему за десятую долю обычного времени. А что, если проблема не сформулирована? В программировании, как и во многих областях, самые большие трудности возникают не с решением проблем, а с выбором решаемых проблем. Воображение трудно измерить, но на практике оно доминирует над производительностью, измеряемой в строках кода.

В любой области существуют различия в производительности, но немало найдется областей, в которых они были бы столь заметными. Различия между программистами настолько велики, что они принимают качественный характер. Впрочем, я не думаю, что это явление присуще только программированию. В любой области технология увеличивает различия в производительности. Думаю, что в программировании это происходит лишь потому, что наша работа в значительной мере зависит от технологии. Но сейчас техника вторгается во все области, так что с течением времени различия будут проявляться все в большем количестве областей. А успех компаний и целых стран будет зависеть от того, как они отнесутся к этому факту.

Если различия в производительности возрастают с внедрением технологии, то вклад самых производительных личностей будет не только непропорционально большим, но и будет расти со временем. При достижении точки, когда 90 % результатов группы создается 1 % ее участников, их снижение производительности до среднего уровня (из-за набега викингов, централизованного планирования или других напастей) приведет к большим потерям.

Если мы хотим добиться от особенно производительных людей максимальной пользы, необходимо понимать их. Какими мотивами они руководствуются? Что

---

[1] В своей книге «Мифический человеко-месяц» («The Mythical Man Month»). — *Примеч. ред.*

им необходимо для выполнения своей работы? Как их узнать? Как убедить их работать на вас? И конечно, главный вопрос — как стать одним из них?

Деньги — не главное

Я знаком с несколькими суперхакерами, поэтому я сел и задумался над тем, что у них есть общего. Вероятно, определяющим качеством следует считать то, что они действительно любят программировать. Обычный программист пишет код, чтобы отработать свою зарплату. Великий хакер готов заниматься программированием для собственного удовольствия, поэтому его искренне радует, когда кто-то готов за это платить.

Говорят, великие программисты безразличны к деньгам. Это не совсем так. Действительно, на самом деле их волнует лишь интересная работа. Но если заработать много денег, дальше можно будет заниматься тем, чем захочешь, и по этой причине хакеров привлекают крупные заработки. А пока им приходится каждый день ходить на работу, они больше думают о том, чем они здесь занимаются, нежели о том, сколько за это платят.

С экономической точки зрения этот факт чрезвычайно важен; он означает, что зарплата великого хакера не обязана даже отдаленно соответствовать его ценности. Великий программист может работать в десятки и сотни раз производительнее обычного, но он будет считать, что ему повезло, если ему будут платить втрое больше. Как я объясню позднее, отчасти это связано с тем, что великие хакеры не понимают, насколько они хороши — но также и потому, что деньги для них не главное.

Что нужно хакеру? Как и любому ремесленнику, ему нужен хороший инструмент. Впрочем, это сказано слишком мягко. Для хорошего хакера работать с плохими инструментами невыносимо. Они просто отказываются работать над проектами с плохой инфраструктурой.

В одной из новых фирм, в которых я работал, на доске объявлений была вывешена реклама от IBM с изображением AS400[1] и подписью: «презирается хакерами»[2].

При выборе инфраструктуры проекта вы принимаете не только техническое решение. Вы также принимаете социальное решение, которое может сыграть более важную роль. Например, если ваша компания хочет написать некоторую программу, может показаться, что ее будет разумно написать на Java. Однако выбор языка также оборачивается выбором сообщества. Программисты, которых вы сможете нанять для работы над проектом Java, будут менее умны, чем те, которых можно нанять для работы над проектом, написанным на Python. А качество хакеров, вероятно, значит больше, чем выбранный язык... Хотя честно говоря, тот факт, что хорошие хакеры отдают предпочтение Python перед Java, кое-что говорит об относительных достоинствах этих языков.

---

[1] Уменьшенная версия мейнфрейма IBM, разработанная для запуска тех же программ на более дешевом оборудовании. Применяется почти исключительно в коммерческих фирмах для скучных коммерческих дел. — *Примеч. ред.*

[2] Справедливости ради должен отметить, что IBM делает хорошее «железо». Я пишу эту статью на портативном компьютере IBM.

Руководство предпочитает более популярные языки, потому что оно рассматривает язык как стандарт. Ему не хочется сделать ставку на технологию, которая уйдет в небытие. Тем не менее, у языков есть одно особое свойство: они представляют собой не только стандарты. Если вам потребовалось переслать биты по сети — пожалуйста, используйте TCP/IP. Но язык программирования не ограничивается форматом, это среда выражения мыслей.

Я читал, что язык Java сменил Cobol в роли самого популярного языка. Для стандарта ничего более и желать нельзя, но для среды выражения существуют гораздо более удачные варианты. Из всех известных мне великих программистов только один готов добровольно программировать на Java, да и тот работает на Sun.

Великие хакеры также обычно настаивают на использовании программ с открытыми исходными кодами. Дело не только в том, что эти программы лучше, но и в том, что они обеспечивают более высокий контроль. Хороший хакер желает контролировать все происходящее. Это один из аспектов того, что делает их хорошими хакерами: если что-то сломалось, они должны это «что-то» починить. Желательно, чтобы они проявляли те же чувства по отношению к тому продукту, который они пишут для вас. Не стоит удивляться тому, что аналогичные чувства распространяются на операционную систему.

Пару лет назад мой знакомый предприниматель рассказал о новой фирме, в создании которой он участвовал. Проект казался многообещающим. Но когда я говорил с ним в следующий раз, он сказал, что программу было решено создавать на платформе Windows NT, и он только что нанял очень опытного разработчика NT начальником технического отдела. Услышав это, я подумал, что фирма обречена. Во-первых, начальник технического отдела не может быть первоклассным хакером; ведь чтобы стать опытным разработчиком NT, ему пришлось добровольно использовать NT, и притом неоднократно, а я не могу себе представить великого хакера, который бы на это пошел. Во-вторых, даже если он и был хорош, ему вряд ли удалось бы нанять кого-нибудь стоящего на проект для NT[1].

Последний рубеж

Вероятно, вторым по важности инструментом хакера после программного обеспечения является его рабочее место. Крупные компании полагают, что главная функция рабочего места — выражение ранга работника. Но хакеры используют свое рабочее место для более важных целей: для них это место, где можно подумать. А ведь в технологических компаниях именно мысли являются основным продуктом. Таким образом, заставлять хакера работать в шумной, отвлекающей обстановке — все равно что пытаться производить краску в помещении, наполненном копотью и пылью.

Газетные комиксы Дилберта часто посвящены офисным секциям, и на то есть веская причина. Все известные мне хакеры ненавидят офисные секции. Одна вероятность того, что тебя прервут, может помешать хакеру работать над сложной

[1] И действительно, компания была обречена. Они закрылись через несколько месяцев.

проблемой. Если вы хотите выполнить настоящую работу в офисе с секциями, возможны два варианта: работа на дому или приход рано/поздно/по выходным, когда на работе больше никого нет. Неужели компании не понимают, что здесь что-то не так? Офис должен быть местом, в котором люди работают, а не местом, создающим максимум сложностей для работы.

Такие компании, как Cisco, гордятся тем, что у них офисная секция имеется у каждого работника — даже у генерального директора. Однако это не такое большое достижение, как они считают; очевидно, офисное пространство в таких компаниях все еще рассматривается как знаки различия. Также следует вспомнить, что компания Cisco известна минимальным объемом внутренних разработок. Они получают новые технологии, скупая новые фирмы, где эти технологии создавались — и где, вероятно, у хакеров было спокойное рабочее место.

Примером крупной компании, понимающей потребности хакеров, служит Microsoft. Однажды я видел рекламу с приглашением на работу в Microsoft, на которой была изображена большая дверь. Работайте на нас, обещала реклам, и мы дадим вам рабочее место, на котором вы сможете выполнять свою работу. И вы знаете, Microsoft отличается от других крупных компаний тем, что она действительно может разрабатывать программное обеспечение своими силами. Не очень хорошо, конечно, но бывает и хуже.

Если компании хотят, чтобы хакеры трудились продуктивно, достаточно посмотреть, как они работают дома. Дома хакер расставляет все по своему вкусу, чтобы сделать как можно больше. Дома хакеры не работают в шумных, открытых местах; они предпочитают комнаты с дверями. Вместо стеклянных коробок, окруженных многочисленными автостоянками, они работают в уютном окружении, где рядом есть соседи и места для прогулок, когда потребуется что-нибудь обдумать. Если хакера одолевает усталость, он вздремнет на диване вместо того, чтобы в коме сидеть за столом и притворяться, будто он работает. Здесь нет уборщиков с пылесосами, ревущими каждый вечер в основные рабочие часы хакера. Здесь нет совещаний, или боже упаси, корпоративных вечеринок и тренировок по развитию командного духа. А если взглянуть, что они делают на своем компьютере, вы увидите, что это лишь подтверждает сказанное ранее об инструментарии. Хакеры могут использовать Java и Windows на работе, но дома, где они пользуются свободой выбора, вы чаще увидите Perl и Linux.

В самом деле, вся эта статистика по поводу Cobol или Java как самого популярного языка может сбить с толку. Если вы хотите знать, какой инструмент лучше всего подойдет для решения, нужно смотреть на другое: что выберет хакер, если предоставить ему полную свободу (иначе говоря, в своем собственном проекте)? Если задать этот вопрос, выясняется, что операционные системы с открытыми исходными текстами уже занимают господствующую долю рынка, а языком «номер один», вероятно, является Perl.

Интересная работа

Наряду с хорошим инструментарием хакеру нужны интересные проекты. Что делает проект интересным? Очевидно, работать над впечатляющими приложениями

(скажем, бортовыми системами самолетов-невидимок или спецэффектами в кино) будет интересно. Однако любое приложение может быть интересным, если в нем приходится решать новые технические проблемы. Поэтому трудно предсказать, какая задача понравится хакеру — ведь некоторые из них интересны только тогда, когда работающие над ними люди открывают что-то новое. До появления компании ITA (занимавшейся разработкой программного обеспечения в Orbitz) люди, работавшие над системами поиска оптимальной стоимости перелета на авиалиниях, вероятно, считали, что это одна из самых скучных задач, которые только можно себе представить. Однако ITA сделала ее интересной, найдя для проблемы более амбициозное определение.

Нечто похожее случилось и с Google. На момент основания Google среди так называемых порталов бытовало общепринятое мнение, что поиск — дело скучное и несущественное. Но парни из Google не считали, что поиск был неинтересным, и поэтому им удалось так хорошо реализовать его.

В этой области многое зависит от руководителя. Хороший руководитель иногда способен переопределить задачу и сделать ее более интересной — как отец, который говорит ребенку: «Спорим, ты не сможешь прибраться в своей комнате за 10 минут». Особенно хорошо это получается у Стива Джобса, отчасти просто за счет высоких стандартов. До появления Mac существовало множество небольших, недорогих компьютеров. Стив переопределил задачу так: сделать такой компьютер, который был бы красивым. И это, вероятно, повлияло на разработчиков сильнее, чем любой кнут или пряник.

Бесспорно, у них это получилось. При первом появлении Mac даже не нужно было включать компьютер, чтобы понять, насколько он хорош; это было видно по упаковке. Несколько недель назад я шел по улице в Кембридже, и вдруг увидел в мусорном контейнере переносную сумку для Mac. Внутри действительно оказался Mac SE. Я отнес его домой, включил в сеть — и он загрузился. Знакомая счастливая рожица Macintosh... Господи, это было так просто. Совсем как... Google.

Хакеры любят работать на людей с высокими стандартами. И все же требовательность — это еще не все. Нужно правильно выбирать то, на чем следует настаивать. А это обычно означает, что руководитель сам должен быть хакером. Мне часто попадаются статьи по поводу того, как управлять программистами. На самом деле таких статей должно быть всего две: для руководителей, которые одновременно являются программистами, и для тех, кто программистом не является. Вторая статья сводится к двум словам: бросайте это дело.

Дело даже не в повседневном руководстве. Действительно хороший хакер в руководстве практически не нуждается. Дело в том, что если вы не являетесь хакером, то и не сможете сказать, кто им является. Аналогичная проблема объясняет уродство американских автомашин. Я называю это «парадоксом дизайна»: казалось бы, чтобы ваши продукты выглядели красиво, достаточно поручить их конструирование хорошему дизайнеру. Но если у вас нет хорошего вкуса, как узнать хорошего дизайнера? Судить об этом по портфолио вы не сможете по определению. Также бесполезно смотреть на премии и предыдущие места работы, потому что в дизайне, как и в большинстве областей, эти факторы определяются скорее модой и сплетнями, а реальные способности уходят на третий план. Получается

заколдованный круг; невозможно управлять процессом, направленным на создание красивых вещей, не умея отличать красивое от безобразного. Американские машины уродливы потому, что ими руководят люди с плохим вкусом.

Многие люди в этой стране считают вкус чем-то эфемерным и даже несерьезным. Конечно, это не так. Чтобы управлять процессом конструирования, руководитель должен быть самым требовательным пользователем продуктов компании. А если у вас, как у Стива Джобса, будет хороший вкус, вы также будете ориентироваться на те задачи, над которыми нравится работать хорошим людям.

Противные мелочи

Довольно легко сказать, какие проекты не являются интересными с точки зрения хакеров: те, в которых вместо решения нескольких больших, четких проблем приходится решать множество мелких. Одним из худших примеров является написание интерфейса к программе, полной ошибок. Другая проблема — настройка чего-то под сложные и плохо определенные потребности клиента. Для хакера такие проекты превращаются в настоящую пытку.

У этих противных мелочей есть отличительная особенность: они ничему не учат. Написать компилятор интересно, потому что в процессе вы узнаете, что такое компилятор. Но написание интерфейса к программе, содержащей много ошибок, ничему не учит, потому что ошибки носят случайный характер[1]. Так что не только привередливость заставляет хороших хакеров избегать противных мелочей — скорее, это вопрос самосохранения. Работа над противными мелочами оглупляет. Хороший хакер избегает ее по тем же соображениям, по которым модели не едят чизбургеры.

Конечно, некоторые проекты изначально имеют такую природу. По закону спроса и предложения они особенно хорошо оплачиваются. Таким образом, компания, которая сможет заставить великих хакеров работать над скучными проблемами, будет особенно успешна. Но как это сделать?

Например, это бывает в новых фирмах (startups). Скажем, в нашей фирме системным администратором работал Роберт Моррис[2]; это примерно то же самое, как «Роллинг Стоунз» на школьной вечеринке. Такие таланты невозможно нанять. Но для компаний, основателями которых они являются, люди готовы выполнять любую черную работу[3].

В крупных компаниях проблема решается методом деления. Для привлечения умных людей создается научно-исследовательский отдел, работникам которого не

---

[1] Видимо, это то, что люди имеют в виду, говоря о «смысле жизни». На первый взгляд идея выглядит странно. Жизнь — не математическое выражение; как она может иметь смысл? Но у нее есть свойство, которое имеет много общего со смыслом. При написании компилятора приходится решать множество проблем, но все эти проблемы относятся к определенной части спектра, как в сигналах. Если же спектр решаемых проблем абсолютно случаен, они воспринимаются как шум.

[2] Роберт Моррис — один из ведущих экспертов в области сетей Unoix. В настоящее время является профессором в MIT. — *Примеч. ред.*

[3] Например, Эйнштейн одно время конструировал холодильники (и имел долю акций).

приходится напрямую работать над противными мелкими проблемами[1]. В этой модели научно-исследовательский отдел работает как шахта. Он выдает не конечный продукт, а новые идеи; возможно, другим работникам удастся из них что-то сделать.

Впрочем, доходить до крайностей не обязательно. Восходящее (bottom-up) программирование открывает другую возможность структурирования: поручите умным людям разработку инструментария. Если компания производит программы для решения некоторой задачи, пусть одна группа создает инструменты для написания программ этого типа, а другая группа использует эти инструменты для написания приложений. При этом умные люди будут писать 99 % кода, но останутся так же изолированными от конечных пользователей, как в традиционных научно-исследовательских отделах. У разработчиков инструментария тоже будут свои пользователи, но их круг будет состоять только из программистов этой же компании[2].

Если бы в компании Microsoft использовали этот подход, ее продукты не содержали бы такого количества дефектов безопасности: менее квалифицированные люди, занимающиеся написанием приложений, обходились бы без низкоуровневых операций вроде выделения памяти. Вместо того, чтобы писать Word на C, они бы занимались сборкой больших готовых блоков.

Группировка

Наряду с интересными проблемами, хакеры также любят общество других хакеров. Великие хакеры склонны собираться в сообщества, как в Xerox Parc. Таким образом, количество привлекаемых хакеров не связано линейной зависимостью с тем, насколько хорошие условия вы для них создадите. Тенденция к группировке означает, что зависимость будет ближе к квадратичной. В любой момент времени найдется только десять-двадцать мест, в которых захотят работать хакеры, и в остальных местах их не просто меньше — их там вообще нет.

Присутствие великих хакеров само по себе еще не гарантирует успеха компании. В Google и ITA, двух самых популярных местах, этот способ отлично сработал, однако он не помог Thinking Machines и ITA. У компании Sun некоторое время дела шли хорошо, но у нее были трудности с бизнес-моделью. В такой ситуации не помогут даже лучшие хакеры.

Впрочем, я думаю, что при равенстве всех остальных факторов компания, способная привлечь великих хакеров, будет обладать громадными преимуществами. Некоторые люди с этим не согласны. Когда мы в 1990-х создавали многочисленные фирмы с рискованным вложением капитала, многие говорили, что успех разработчиков программного обеспечения зависит не от написания хороших программ, а от известности брэнда, каналов доминирования и заключения удачных сделок.

---

[1] Трудно сказать, что именно следует считать аналогом научных исследований в компьютерных технологиях. В первом приближении это программа, не имеющая пользователей.

[2] Похожая схема уже давно используется в строительной отрасли. Двести лет назад строители возводили весь дом от начала и до конца. Но постепенно работа строителей все в большей степени сводится к сборке компонентов, сконструированных и выпущенных кем-то другим. Как и появление настольных систем компьютерной верстки, эта схема дает людям свободу для разрушительных экспериментов, но ее эффективность сомнений не вызывает.

Похоже, они действительности верили этому, и я знаю, почему. Видимо, многие фирмы такого рода (по крайней мере подсознательно) рассчитывают повторить успех Microsoft. И конечно, если вы берете Microsoft за образец, незачем искать компании, надеющиеся выиграть за счет написания хороших программ. Но надежды на появление «новой Microsoft» ошибочны; ни одна новая фирма не станет «новой Microsoft», если другая компания не прогнется в нужный момент, чтобы стать «новой IBM».

Использовать Microsoft в качестве модели неверно, потому что весь ее успех обусловлен одним счастливым случаем. Microsoft — аномальная точка выборки. Если исключить ее, выясняется, что на рынке обычно побеждают хорошие продукты. Финансистам следует ориентироваться на новые Apple или Google.

Думаю, Билл Гейтс знает об этом. В Google его беспокоит не столько известность брэнда, сколько присутствие лучших хакеров[1].

Идентификация

Итак, кто же эти великие хакеры? Как узнать их при встрече?

Оказывается, это очень трудно. Даже сами хакеры не знают этого. Я убежден, что мой друг Тревор Блэкуэлл (Trevor Blackwell) является великим хакером. Возможно, вы читали на Slashdot, что он собрал собственный скутер Segway[2]. Самое замечательное в этом проекте то, что все программное обеспечение Тревор написал за один день (кстати говоря, на Python) и бормотал что-то невнятное. Помню, я стоял у него за спиной и делал отчаянные жесту Роберту, чтобы он выставил этого психа из офиса, и мы могли пойти обедать. Роберт говорит, что он сначала тоже недооценил Тревора. Когда они встретились впервые, Тревор только что придумал новую схему наведения порядка, в которой каждый аспект его жизни записывался на специальных учетных карточках. Все эти карточки он постоянно носил с собой. Кроме того, он только приехал из Канады и говорил с сильным акцентом.

Проблема осложняется еще одним обстоятельством: принято считать, что хакеры равнодушны к общественному мнению, но иногда они очень любят казаться умными. В свою бытность аспирантом я часто заходил в лабораторию искусственного интеллекта MIT. На первый взгляд это было довольно жуткое место — все говорили так быстро... Но вскоре я научился этому фокусу. Думать быстрее для этого не нужно; достаточно использовать вдвое больше слов.

При таком уровне шума в сигнале хорошего хакера довольно трудно узнать при первой встрече. Я так и не умею это делать, даже сейчас. Также нельзя судить

---

[1] Google представляет для Microsoft гораздо большую опасность, чем Netscape. Возможно, Google — самый опасный конкурент Microsoft за все время существования компании. Не в последнюю очередь это объясняется готовностью сражаться. На странице с перечнем вакансий Google говорится, что одним из основных качеств нового работника должно быть «отсутствие склонности к злу». От компании, продающей растительное масло или горнодобывающее оборудование, такое заявление смотрелось бы эксцентрично. Однако всему компьютерному сообществу должно быть ясно, кому адресовано это объявление войны.

[2] См. http://www.tlb.org/scooter.html — *Примеч. перев.*

о качестве хакера по резюме. Похоже, единственный способ оценить хакера — поработать с ним над чем-нибудь.

И это одна из причин, по которым высокотехнологические зоны часто возникают рядом с университетами. Активным ингредиентом здесь являются не столько профессора, сколько студенты. Новые фирмы растут рядом с университетами, потому что университеты сводят вместе перспективных молодых людей и заставляют их работать над одними проектами. Умные студенты находят других умных студентов и начинают совместно создавать свои собственные проекты.

Поскольку великого хакера можно узнать только в ходе совместной работы, сами хакеры не могут оценить себя. Впрочем, этот принцип до определенной степени справедлив во многих областях. Оказывается, люди, хорошо разбирающиеся в чем-то, не столько убеждены в собственном величии, сколько озадачены некомпетентностью окружающих.

Но для хакера самооценка особенно сложна, потому что им трудно сравнивать свою работу. В других областях это проще. В стометровке самый быстрый выявляется за 10 секунд. Даже в математике существует общее мнение относительно того, какие задачи особенно сложны, и какое решение следует признать хорошим. Но хакерство сродни писательскому труду. Кто может сказать, какой из двух романов лучше? Уж конечно, не их авторы.

Правда, в среде хакеров можно прислушаться к мнению других хакеров. Дело в том, что, в отличие от писателей, хакеры совместно работают над проектами. Подбросив своему коллеге несколько трудных задач, вы довольно быстро узнаете, насколько сложны они были. Но хакеры не могут наблюдать за собой во время работы. Поэтому если спросить великого хакера, насколько он хорош, он почти наверняка ответит: «Я не знаю». И это не проявление скромности — он действительно не знает. И никто не знает, кроме тех людей, с которыми он работал. Возникает странная ситуация: мы не знаем своих героев. Хакеры становятся знаменитыми вследствие случайных происшествий или пиара. Время от времени бывает нужен пример великого хакера, и я не знаю, кого использовать. Первыми приходят в голову имена людей, которых я знаю лично, но использовать их кажется неудобно. Может, назвать Ричарда Столлмена, Линуса Торвальдса, Алана Кея или другую знаменитость? Но я понятия не имею, являются ли они великими хакерами. Я никогда не работал с ними ни над одним проектом. Даже если среди хакеров найдется свой Майкл Джордан, об этом никто не знает, включая его самого.

Становление

Напоследок остается вопрос, который задают себе все хакеры: как стать великим хакером? Не знаю, возможно ли это. Но несомненно, существуют вещи, от которых люди глупеют; а если можно сделать себя глупее, то наверное, возможно и обратное.

Возможно, ключом к становлению хорошего хакера является работа над тем, что вам нравится. Вспоминая известных мне великих хакеров, я могу выделить у них единственную общую черту: их крайне сложно заставить работать над тем, над чем они не хотят работать. Не знаю, что это — причина или следствие; возможно, и то, и другое. Чтобы хорошо справиться с какой-то работой, ее нужно

любить. Так что скорее всего вы сможете хорошо справляться с программированием в той степени, в которой вы его любите. Попробуйте вспомнить чувство восхищения, которое у вас вызывало программирование в 14-летнем возрасте. Если вас беспокоит, что от текущей работы у вас засыхают мозги, скорее всего, так оно и есть. Конечно, лучшие хакеры обычно умны, но это относится ко многим областям. Существует ли качество, присущее только одним хакерам? Я спрашивал своих друзей, и на первое место они поставили любопытство. Я всегда предполагал, что умные люди любопытны — и любопытство является первой производной от знания. Но очевидно, хакеры отличаются особенным любопытством, особенно к тому, «как это работает» и «что у него внутри». И это вполне разумно, потому что программа фактически представляет собой большое описание сути вещей. Некоторые друзья упоминали умение хакера концентрироваться — их способность, как это было сформулировано, «отфильтровать все, что находится за пределами их голов». Конечно, я это замечал — и слышал от нескольких хакеров, что даже после половины кружки пива они совершенно не могут программировать. Так что возможно, хакер также должен уметь сосредотачиваться на решаемой задаче. Возможно, великий хакер способен держать в голове большой объем контекста, чтобы при взгляде на строку кода он видел не только эту строку, но и всю программу. Джон Макфи (John McPhee) писал, что успех Билла Брэдли (Bill Bradley) в баскетболе отчасти объяснялся его выдающимся периферийным зрением. При «идеальном» зрении человек угол вертикального периферийного зрения составляет около 47 °. У Билла Брэдли он был равен 70 °; глядя на пол, он мог видеть корзину. Возможно, великие хакеры обладают похожей врожденной способностью.

Кстати, это также объясняет нелюбовь хакеров к офисным секциям. Администраторы, концентрации которых ничего не мешает из-за отсутствия таковой, и не подозревают, что при работе в офисной секции хакер чувствует себя так, словно его мозги засунули в блендер (кстати, Билл, если слухи о его аутизме верны, хорошо знает об этом на собственном опыте).

Одно из отличий великих хакеров от умных людей вообще состоит в том, что хакеры менее склонны к политкорректности. Если между хакерами существует некий аналог «тайного рукопожатия», так это способность в беседе с хорошо знакомым человеком свободно выражать мнения, за которые на публике их бы забили камнями. И я понимаю, почему политическая некорректность бывает полезным качеством в программировании. Программы очень сложны, и по крайней мере в руках хорошего программиста, чрезвычайно гибки. В таких условиях может пригодиться привычка ставить под сомнение общепринятые истины.

Можно ли воспитать в себе эти качества? Не знаю. Но по крайней мере, можно не подавлять их в себе. Вот вам моя лучшая попытка сформулировать рецепт. Если и возможно воспитать в себе великого хакера, заключите с собой сделку: никогда не работайте над скучными проектами (если только ваша семья не умирает с голоду), и взамен никогда не позволяйте себе халтуру в работе. Все известные мне великие хакеры следовали этим правилам — хотя скорее всего, у них просто не было выбора.

Благодарю Джессику Ливингстон (Jessica Livingstone), Роберта Морриса (Robert Morris) и Сару Харлин (Sarah Harlin) за знакомство с ранними версиями моего доклада.

Джон Грубер

Поле адреса как новая командная строка

Прежде одна мысль о переносе разработки приложений в Web выводила меня из себя. Интерфейсы на базе веб-браузера казались мне огромным шагом назад, данью поколению мэйнфреймов с их кошмарными интерфейсами. Пользователь получает форму, заполняет ее, нажимает Enter и ждет несколько секунд, пока находящийся невесть где мэйнфрейм не решит, какую форму нужно заполнять следующей.

Аааааааа!

Но я изменил свое мнение. Браузеры стали гораздо лучше, все обновилось, а сообщество веб-разработчиков перестало беспокоиться о бедной старой тетушке Мардж, которая все еще работает с Netscape 0.9. Программисты с творческим складом ума демонстрируют потрясающе интерактивные интерфейсы на базе Web — такие, как Flickr (http://www.flickr.com) и Google Maps (http://maps.google.com); при помощи Flash и Ajax (Asynchronous JavaScript and XmlHttpRequest) они создают веб-страницы, возможности взаимодействия с которыми не ограничиваются отправкой формы и ожиданием новой веб-страницы.

Что касается интерактивности, выяснилось, что перетаскивание и быстрые клавиатурные интерфейсы не играют важной роли. Перетаскивание было абсолютно неочевидным, а прежде чем начать, пользователю приходилось по 25 минут размещать окна. Рядовому пользователю было гораздо проще щелкнуть на команде меню, нежели запоминать клавиатурные сокращения. Оказалось, что приложения на базе Web пользуются гораздо большей популярностью, чем — на первый взгляд более эргономичные — интерфейсы Windows.

Компании Microsoft это не сулит ничего хорошего. К ее чести следует заметить, что она заметила это гораздо раньше других — еще в те времена, когда Марк Андрисен (Marc Andreesen) называл Windows «набором плохо отлаженных драйверов устройств», и никто не понимал, что это означало. А выбранная Microsoft стратегия борьбы была настолько смелой, что ее можно было сравнить с жертвой ферзя в шахматах: компания должна была создать собственный браузер, сделать его настолько лучше других, чтобы все им пользовались, подавить конкуренцию — но не настолько, чтобы браузер мог заменить Windows при разработке приложений с расширенной функциональностью. Это был очень смелый ход, который работал в течение трех лет, однако в конечном счете плетью

обуха не перешибешь; программисты начали мыслить творчески, к этому както примешался объект XmlHttpRequest, а Microsoft переоценила привязанность среднего пользователя к маленьким удобствам пользовательского интерфейса Windows. — *Ред.*

Авторам, регулярно публикующим свое мнение, бывает трудно устоять перед искушением похвастаться оправдавшимся прогнозом. Также довольно легко забыть о тех случаях, когда прогнозы не сбылись. К счастью, в середине и в конце 1990-х годов я еще не публиковал статьи о разработке программного обеспечения. «К счастью» — потому что иначе мне сейчас пришлось бы сильно комплексовать по поводу того, что я написал бы о потенциале Web как платформы разработки приложений.

В то время война браузеров Netscape — Microsoft была в самом разгаре, и общественное мнение считало, что в Web скрыто будущее разработки приложений. Конечно, технология еще не существовала, но народ считал, что браузер Netscape представляет серьезную угрозу для монополии Microsoft с системой Windows — в какой-то момент пользовательские приложения будут разрабатываться в расчете на запуск из браузера.

Отсюда и невероятная смена курса Microsoft, от более или менее полного игнорирования Интернета до полного доминирования за несколько ближайших лет. Считалось, что Microsoft погубила компанию Netscape, потому что увидела в ней угрозу Windows.

Впрочем, лично я не попался на эту удочку. Я в полной мере осознавал потенциал Web как среды публикации, но не видел, каким образом Web сможет когдалибо выступить в роли среды разработки высококачественных приложений. Мне казалось, что Microsoft погубила Netscape не из-за угрозы для Windows, а просто потому, что они (Microsoft) хотели полностью контролировать новую среду публикации.

Трудно было ошибиться сильнее. В сущности, общественное мнение оказалось правым — Web действительно превратилась в популярную среду разработки приложений. Моя ошибка заключалась в предположении, что непременным условием популярности Web должны стать высокие стандарты качества приложений.

Я мыслил в контексте приложений, использовавшихся в моей повседневной работе около 1996 года: BBEdit, QuarkXPress, Photoshop, Eudora. В то время «веб-приложение» ни при каких условиях не могло обеспечить такого качества, как «настоящие» приложения. И в этом я был прав — программы, работающие в браузере, уступали по удобству и интерактивности.

Но при этом я упустил из виду тот факт, что большинство людей не пользуется сложными редакторами текстов и системами компьютерной верстки; что еще важнее, они часто вообще не думают об удобстве и интерактивности. Совсем. Они просто хотят, чтобы приложение работало, и было по возможности «простым».

Мое мнение о том, что веб-приложения никогда не станут популярными, напоминало мнение театральных критиков начала 1950-х годов, отвергавших телевидение.

Ограничения веб-приложения с точки зрения пользователя очевидны. Такие приложения попросту выглядят и работают не так, как обычные настольные

приложения. Браузер, в котором они выполняются, — да, это нормальное приложение. Но веб-приложения, работающие в браузере, нормальными назвать нельзя. У них нет меню и комбинаций клавиш (которые есть у самого браузера). Дело даже не в пресловутом «стиле Mac» — проблема в равной степени относится к Windows и настольным платформам с открытыми кодами. Веб-приложения воспринимаются как веб-страницы, а не как настоящие приложения для настольных систем Mac/Windows/Linux.

Мелкие аспекты пользовательского интерфейса, которым я уделяю столько внимания — ничто по сравнению с капитальными недостатками даже самого лучшего веб-приложения. Но большинству людей до этого нет дела, потому что веб-приложения так чертовски просты в использовании. Здесь интересно то, что веб-приложения «просты», несмотря на свои очевидные упущения в области пользовательского интерфейса.

К преимуществам из области простоты использования следует отнести то, что веб-приложения не нуждаются в установке, а пользователю не нужно беспокоиться о том, где и как будут храниться данные.

Образец А: веб-приложение для работы с электронной почтой. В отношении функциональности, особенно вспомогательных возможностей вроде отполированного пользовательского интерфейса, поддержки перетаскивания и широкого набора комбинаций клавиш, почтовые клиенты на базе Web не могут сравниться с обычными клиентами для настольных систем.

Но...

Работая с веб-клиентом, вы можете принять электронную почту в любом браузере с любого компьютера, подключенного к Интернету. Процесс «установки» сводится к вводу URL в поле адреса. В сущности, поле адреса превращается в новую командную строку.

Сервис Google Gmail поднял стандарты на новый уровень и предоставил ряд возможностей, которые достойно выглядят на фоне традиционных почтовых клиентов — это и быстрый, точный поиск (разумеется), и очень удобная система отображения сообщений с делением на темы. Кроме того, Gmail поддерживает комбинации клавиш, реализованные на JavaScript; но как их описывает Марк Пилгрим (Mark Pilgrim) в своем обзоре Gmail[1], они «[выглядят так, словно] были спроектированы пользователями vi[2] (*j* — перемещение вниз, *k* — перемещение вверх, и пользователю предлагается запоминать последовательности из нескольких клавиш)».

Тематическое отображение сообщений и средства поиска в Gmail действительно хороши, но по общему впечатлению Gmail значительно уступает «настоящему» почтовому клиенту для настольных систем. К «плюсам» следует отнести общие особенности всех почтовых веб-приложений — отсутствие установки, отсутствие сопровождения, доступ с любого компьютера (в том числе и с работы — немаловажный фактор для персональной электронной почты). Gmail попросту лучше

---

[1] См. http://diveintomark.org/archives/2004/04/10/gmail-accessibility.

[2] Древний текстовый редактор для Unix; создавался еще до широкого распространения клавиш управления курсором. — *Примеч. ред.*

любого почтового веб-приложения, однако Yahoo, Hotmail и другие все еще продолжают пользоваться бешеной популярностью.

Отвергая веб-приложения десять лет назад, я не понимал одного: веб-приложения вовсе не обязаны победить настольные приложения в борьбе «на равных». Соревнование ведется на совершенно иных условиях. Вы просто вводите URL, и получаете доступ к своей электронной почте. Из этого следует все остальное.

Кто проиграет, если победят веб-приложения?

На эту тему меня натолкнул великолепный очерк Джоэла Спольски «Как Microsoft проиграла войну API»[1], опубликованный на прошлой неделе. Суть очерка Спольски заключается в том, что главной драгоценностью Microsoft был Win32 API — совокупность программных интерфейсов, используемых разработчиками при написании Windows-приложений для настольных систем, и что разработка веб-приложений набирает темпы непосредственно за счет разработки Win32.

Почему же Win32 API так важен для Windows-монополии Microsoft? Дело в зависимости: если ваша компания зависит от программного обеспечения Win32, она также зависит от Windows. И наоборот, с точки зрения разработчика программирование для Win32 API обеспечивает работу программы на 90 % компьютеров в мире. Возникает замкнутый круг, который принес своим создателям 50 миллиардов долларов — клиенты используют Windows, потому что для этой системы пишутся программы, а разработчики программируют для Windows, потому что здесь они находят клиентов.

Переход на другую платформу — скажем, на Mac OS X — обойдется крупной корпорации очень дорого. Потребуется не только полная смена всего оборудования, но и полная смена всего программного обеспечения. Причем речь идет не только о покупке новых лицензий — крупным корпорациям также придется заменять все специальные приложения, написанные внутренними разработчиками (а как вы думаете, чем столько лет занимались все эти программисты Visual Basic?)

Переход на настольные системы с открытыми кодам — KDE, Gnome и др. — тоже недешев. Нет, новое оборудование не понадобится, но проблемы с программным обеспечением остаются. (Да, я в курсе — приложения Win32 можно запускать в Linux при помощи эмулятора Win32 Wine[2], и на Mac при помощи Virtual PC, но эти среды Win32 являются «второсортными». Я не говорю, что это невозможно; просто этот вариант непривлекателен.)

Однако переход на веб-приложения — совсем другое дело. Он может осуществляться постепенно; вы заменяете одно приложение за другим, продолжая работать в Windows, причем при этом продолжают работать все остальные программы Win32.

---

[1] Из книги «Joel on Software», Apress 2004. Очерк также доступен в Интернете по адресу http://www.joelonsoftware.com/articles/APIWar.html.

[2] См. http://www.winehq.com/.

Переход на веб-приложения не только дешев, но и прост. Более того, во многих отношениях переход работников на веб-приложения происходит проще, чем с обновлением уже используемых приложений Win32. Иначе говоря, корпорациям проще понемногу переходить на веб-приложения, чем ограничиваться одними Windows-приложениями.

Процесс развертывания веб-приложений также проще обычного. Программы не обязательно устанавливать на каждом клиентском компьютере; приложение существует в единственном экземпляре на веб-сервере. Каждый пользователь автоматически получает новейшую версию программы. Специализированные веб-приложения также превосходят традиционные приложения по простоте разработки. Это не означает, что разработчик может легко создать веб-приложение, неотличимое от традиционного приложения — более того, это невозможно. Зато он может легко написать приложение, схожее с веб-страницей, и этого вполне достаточно для многих целей — особенно в приложениях ввода и выборки данных, подключающихся к базам данных SQL на серверах.

И если 90 % компьютеров, на которых могут работать приложения Win32, составляют большую долю рынка — то на каком количестве компьютеров можно запустить типичное веб-приложение?

Большинство почтовых веб-приложений (например, Gmail или Yahoo Mail) работает на любом компьютере с Internet Explorer, Safari или любом браузере семейства Mozilla. Большинство веб-приложений для работы с блогами (такие, как Blogger, Movable Type, Wordpress и TextPattern) также работали во всех опробованных мною браузерах. Фактически они запускаются на любом компьютере, подключенном к Интернету.

Я уже давно размышляю о становлении Web как платформы разработки приложений. Но до того момента, когда я на прошлой неделе прочитал очерк Спольски, мне не приходил в голову один любопытный факт: компания Microsoft капитально ошиблась, выбрав своей мишенью Netscape. Угрозу Windows как платформе разработки приложений представляла вовсе не Netscape, а Web как таковая.

Они потратили массу времени, денег и усилий на разработку IE, добились монополии и сокрушили Netscape — и что толку? Web продолжает завоевывать умы разработчиков в ущерб Win32.

Разумеется, существуют исключения (на ум приходят банковские сайты), и все же большей частью веб-приложения рассчитаны на работу в любом современном браузере, не только в IE.

Полагаю, Спольски абсолютно прав в том, что Microsoft проигрывает войну API. Но какая ирония судьбы — Microsoft проигрывает эту войну, несмотря на свою победу в войне браузеров. А ведь победа в войне браузеров (и уничтожение Netscape) должна была раз и навсегда предотвратить саму возможность войн API.

Грегор Хопе

очему в Starbucks не используют двухфазное закрепление

Много лет назад я работал на провайдера, развертывавшего новый сервис DSL. Конечно, этот провайдер не смог бы предоставить услуги DSL самостоятельно. Для этого потребовались бы машины, техники, провода и тому подобные неприятные вещи. Мой работодатель просто перепродавал сервис DSL, предоставляемый компанией Covad, которая в свою очередь сильно зависела от сотрудничества с местными телефонными компаниями... которые в свою очередь более всего на свете хотели бы, чтобы компания Covad умерла, умерла, умерла!!!

Как бы то ни было, меня больше всего поразила невероятно запутанная процедура заключения контракта. Наш отдел обслуживания клиентов вывесил громадный плакат с гигантской, неимоверно усложненной блок-схемой. Взглянув на блок-схему с точки зрения программиста, я нашел в ней множество очевидных дефектов... «Багов», на жаргоне программистов. Специалисты, разрабатывавшие процедуру подключения, были хорошими людьми, но не программистами. Никто не рассказывал им про обработку исключений, асинхронные операции, двухфазное закрепление и многопоточность, и даже о простейшей команде if. Название этой статьи немедленно бросилось мне в глаза. Я как программист довольно часто оцениваю сценарии повседневной жизни в контексте программирования и метафор программной архитектуры.

Мне кажется, что следует обратить больше внимания на проведение параллелей между архитектурой бизнес-процессов и архитектурой программного обеспечения. Мне кажется, современный бизнес настолько сложен, что для его правильного ведения необходимо обладать квалификацией проектировщика программного обеспечения. В частности, мне абсолютно ясно, что кошмарный уровень сервиса во многих авиакомпаниях и операторах сотовой связи напрямую обусловлен одним фактом: люди, разрабатывавшие процедуру обслуживания клиентов, попросту не понимают концепцию двухфазного закрепления транзакций. Поэтому вам приходится 15 минут висеть на телефоне, тратить 25 минут на общение с низкоуровневым болваном, который ничем вам помочь не может, потом он наконец соглашается переключить вас на своего начальника, и... щелк, гудки в трубке. Тот, кто спроектировал эту процедуру, не поймет двухфазное закрепление, даже если оно ударит его по голове. — *Ред.*

Хотто какао-о кудасай

Я только что вернулся из 2-недельной поездки в Японию. Мне бросилось в глаза огромное количество кофеен Starbucks® (スターバックス)[1], особенно в районах Синдзюку и Роппонги[2]. В ожидании своей порции горячего какао («хотто какао»), я начал размышлять о том, как в Starbucks организована обработка заказов на напитки. Компания Starbucks, как и большинство других коммерческих организаций, заинтересована прежде всего максимальной пропускной способностью по обработке заказов — чем больше выполнено заказов, тем больше заработок. Вот почему они используют асинхронную обработку: при оформлении заказа кассир помечает чашку вашим заказом и помещает ее в очередь. Очередь буквально представляет собой линию кофейных чашек на эспрессо-машине. Наличие очереди позволяет разделить функции кассира и бариста, а кассир продолжает принимать заказы даже в том случае, если бариста в настоящий момент занят. Кроме того, при большом наплыве клиентов эта схема позволяет задействовать несколько бариста в сценарии «Конкурирующих клиентов»[3].

Корреляция

Наряду с преимуществами асинхронной обработки Starbucks также приходится иметь дело с некоторыми проблемами, изначально присущими асинхронным механизмам. Для примера возьмем корреляцию. Заказы на напитки не всегда выполняются в порядке их размещения. Это может произойти по двум причинам. Во-первых, несколько бариста могут выполнять заказы на разном оборудовании. Например, приготовление коктейлей обычно занимает больше времени, чем приготовление обычного эспрессо. Во-вторых, бариста может приготовить несколько напитков за один заход, чтобы оптимизировать процесс выполнения. В результате у Starbucks возникают проблемы с корреляцией. Напитки доставляются с нарушением исходной последовательности, и их необходимо как-то связать с нужным клиентом. В Starbucks для решения этой проблемы применяется решение, часто встречающееся в архитектурах с сообщениями — корреляционные идентификаторы[4]. В США в большинстве кофеен Starbucks используются прямые корреляционные идентификаторы — имя клиента пишется на чашке, и бариста

---

[1] Starbucks — зарегистрированный товарный знак Starbucks U.S. Brands, LLC.

[2] Если вы никогда не бывали в кофейнях Starbucks, объясню, как они работают. Вы заходите и становитесь в очередь к кассе. Вы точно описываете кассиру ту разновидность кофе, которая вам нужна. Существуют миллионы миллионов всевозможных комбинаций, и вы можете выбрать практически любой аспект своего кофе, что отчасти объясняет популярность заведения. Кассир переводит ваш заказ на стандартную терминологию Starbucks и передает его бариста, то есть человеку, который непосредственно готовит кофе. Бариста берет чашку подходящего размера и записывает заказ на чашке; если бариста очень занят, это делает кассир. Вы платите кассиру и ждете, пока бариста приготовит ваш напиток. Бариста ставит напиток на стойку и называет либо ваше имя, либо описание напитка. — *Примеч. ред.*

[3] См. http://www.eaipatterns.com/CompetingConsumers.html.

[4] См. http://www.eaipatterns.com/CorrelationIdentifier.html.

называет его, когда напиток будет готов. В других странах корреляция осуществляется по типу напитка.

Обработка исключений

Обработка исключений в асинхронных сценариях с сообщениями иногда оказывается трудной задачей. Если критерием истины является практика, возможно, нам удастся узнать что-нибудь полезное, наблюдая за обработкой исключений в Starbucks. Что делает кассир, если клиент не может расплатиться? Он выливает напиток, если он уже был приготовлен, или чашка исключается из очереди. Если напиток приготовлен неправильно или неудовлетворительно, его готовят заново. Если эспрессо-машина ломается, и приготовить напиток невозможно, клиенту возвращают деньги. Каждый из перечисленных сценариев представляет конкретную стратегию обработки ошибок:

Аннулирование. Самая простая стратегия обработки ошибок: не делать ничего или ликвидировать уже проделанную работу. На первый взгляд такое решение выглядит сомнительно, но в повседневной коммерческой деятельности оно может оказаться приемлемым. Если потери невелики, может оказаться, что реализация исправления ошибки обойдется дороже, чем простое аннулирование. Например, я работал на некоторых провайдеров Интернета, которые выбирали такой подход при наличии несоответствий в циклах выставления счетов/обслуживания. В результате клиент может пользоваться услугой без выставления счета. Потери доходов были достаточно малыми, что оправдывало такой подход к ведению бизнеса. Время от времени проводилась процедура согласования, в ходе которой выявлялись и закрывались «бесплатные» учетные записи.

Повторная попытка. Если некоторая группа операций с большим набором («транзакция») завершается неудачей, фактически возможны два варианта: отменить уже внесенные изменения или попытаться повторить те из них, которые не удалось провести. Повторная попытка является приемлемым вариантом при наличии реальных шансов на ее успешное завершение. Например, если операция нарушает действующие бизнес-правила, вряд ли ее удастся повторить со второй попытки. Однако если внешняя система временно недоступна, возможно, повторная попытка окажется успешной. Особым случаем является повторная попытка с Идемпотентным получателем[1]. В этом случае можно просто повторять все операции снова и снова, потому что в случае успеха получатель будет игнорировать дубликаты.

Компенсация. Наконец, в последнем варианте уже завершенные операции отменяются с целью возврата системы в целостное состояние. Например, такие «компенсации» хорошо работают в финансовых системах, где снятые со счета суммы могут компенсироваться их повторным занесением.

---

[1] См. http://www.eaipatterns.com/IdempotentReceiver.html.

Кроме всех перечисленных стратегий, существует механизм двухфазного закрепления с отдельными фазами подготовки и выполнения. В примере с кофейней Starbucks аналогом двухфазного закрепления было бы ожидание клиента у кассы с чеком и деньгами до того момента, когда напиток будет готов. Напиток включается в заказ, после чего деньги, чек и напиток моментально переходят из рук в руки. Ни кассир, ни клиент не могут отлучиться до момента завершения «транзакции». Использование двухфазного закрепления означало бы верную смерть для бизнеса Starbucks, потому что оно привело бы к радикальному снижению количества клиентов, обслуживаемых за некоторый временной интервал. Этот факт лишний раз напоминает нам о том, что двухфазное закрепление существенно упрощает жизнь, но оно же мешает свободной передаче сообщений (а следовательно, ухудшает масштабируемость) из-за необходимости захвата ресурсов транзакций с состоянием в потоке асинхронных операций.

Диалог

Взаимодействие в кофейне также является хорошим примером простого, но распространенного шаблона Диалога[1]. Общение между двумя сторонами (клиент и кофейня) состоит из короткого синхронного взаимодействия (заказ и оплата) и более долгого асинхронного взаимодействия (приготовление и получение напитка). Диалоги такого рода широко распространены в сценариях с приобретением товаров. Например, при размещении заказа на Amazon при коротком синхронном взаимодействии присваивается номер заказа, а все последующие действия (снятие средств по кредитной карте, упаковка, отправка) выполняются асинхронно. После завершения дополнительных операций клиент получает оповещение по электронной почте (также асинхронно). При возникновении каких-либо проблем Amazon обычно применяет компенсацию (возврат средств на кредитную карту) или повторную попытку (повторная отправка потерянных товаров).

Архитектуры в реальной жизни

Подводя итог, можно сказать, что реальный мир часто проявляет свою асинхронную природу. Наша повседневная жизнь состоит из множества скоординированных, но асинхронных взаимодействий (чтение и ответ на сообщения электронной почты, покупка кофе и т. д.). Таким образом, асинхронные архитектуры с сообщениями часто обеспечивают естественную модель для взаимодействий такого рода. Это также означает, что для проектирования успешных решений, основанных на обмене сообщениями, часто бывает полезно поискать аналоги в повседневной жизни. *Домо аригато годзаимас*![2]

---

[1] См. http://www.eaipatterns.com/ramblings/09_correlation.html.

[2] Большое спасибо (японск.). — *Примеч. перев.*

Рон Джеффрис

Страсть

Я работаю программистом, потому что это интересно, и мне нравится этим заниматься — а не потому, что за программирование особенно хорошо платят, или мне нечем убить время до того дня, когда я умру. Мне понравилась статья Рона Джеффриса, посвященная первым волнующим впечатлениям от экстремального программирования.

Полагаю, большая часть этого волнения уже прошла. Экстремальное программирование превратилось в отрасль промышленности со своими дорогостоящими консультантами, докладчиками и книгами, выходящими каждый месяц. Оно также отчасти закостенело: то, что начиналось как поиски истины с ворохом идей (одни получше, другие похуже), превратилось в жесткий канон обязательных практик — как хороших, так и крайне плохих. Сегодня многие программисты, заявляющие о своей приверженности экстремальному программированию, на самом деле рассматривают его, как ресторанное меню: они выбирают пункты меню, которые им нравятся (Нет документации! Класс!), и игнорируют то, что им не по вкусу (разработка, управляемая тестированием; парное программирование). В результате возникает то явление, которое мы традиционно называем «программированием навскидку» — и которое, как все согласятся, Не Работает.

И все же приятно вспомнить старые добрые времена и раннюю весну технологической революции. И я благодарен Рону, которому удалось запечатлеть мимолетное чувство восторга и страсти по поводу программирования. — *Ред.*

Брайан Марик (Brian Marick) в своем блоге[1], который сам по себе весьма интересен, дает ссылку на весьма провокационную статью, посвященную групповому мышлению, немецкой философии и шоу Saturday Night Live[2]. Знакомство с этой статьей заставило меня задуматься об экстремальном программировании и «гибких» (Agile) технологиях.

---

[1] См. http://www.testing.com/cgi-bin/blog/2004/10/03#testing-metaphor.

[2] См. http://www.gladwell.com/2002/2002_12_02_a_snl.htm.

Дедушка рассказывает

Помню те времена, когда экстремальное программирование и гибкие технологии были совсем новым и модным явлением. В то же время их корни уходили в прошлое, к тем временам, когда мы писали замечательные программы и реагировали на потребности наших клиентов, проявляя фантастическое воображение и командный дух. Все хорошее из наших лучших дней оставалось с нами, хотя было и много нового — что отчасти компенсировало то плохое, которое всегда является неизбежной частью хорошего.

Многие из нас писали об этих идеях. Мы собирались вместе, чтобы поговорить, поспорить и подумать о том, что происходит вокруг и как это может изменить мир. Одно время мы даже задумали написать книгу; ее авторами должны были стать Кент Бек (Kent Beck), Уорд Каннингем (Ward Cunningham), Мартин Фаулер (Martin Fowler) и Кен Ауэр (Ken Auer). Они даже собирались включить меня в коллектив авторов.

Мы встречались со всеми тогдашними авторитетами в области гибких методологий и множеством других людей, заинтересованных и полных энтузиазма. Новые люди появлялись, вносили свой вклад, объясняли, задавали вопросы и развивали идеи.

Состоялась знаменитая Сноубердская конференция, на которой мы написали Манифест гибких методологий[1]. Только взгляните на список имен; великие люди несколько дней оставались в одной комнате, формируя понимание тех принципов, над которыми мы думали. Мы развивали мысли друг друга, спорили между собой. Это было чудесно.

То, что мы сделали, оказало большое влияние на мир разработчиков. Появилось немало великих умов; одни приехали в Сноуберд, другие присоединились к нам позже. Движение породило множество книг, семинаров на конференциях и даже собственные международные конференции. У нас были свои группы новостей и списки рассылки. Даже появились серьезные недоброжелатели, объяснявшие, что наши идеи слишком радикальны, в принципе неработоспособны, а мы то ли ненормальные, то ли злодеи. Странное было время.

Но как было здорово! В нас бушевал настоящий огонь — волнующий, возвышенный, энергичный. У нас была миссия и своя доля истины.

Рожденные для страсти

Я был рожден для страсти — страсти к своей работе и к людям, связанным с ней. Мне успешно удавалось формировать команды, справляющиеся со своими задачами, хотя не обходилось и без изрядных неудач. Одни люди любили меня, другие ненавидели, и хотя я определенно предпочитаю первое, оба варианта все же лучше равнодушия. Потому что я не хочу оставлять людей равнодушными; я хочу что-то изменить.

---

[1] См. http://www.agilemanifesto.org

Вот в чем на мой взгляд заключается суть этого движения: в желании что-то изменить.

И вот чего я от него добиваюсь: чтобы оно что-то изменяло.

Вот каким я пытаюсь быть, и какие качества мне хотелось бы видеть в окружающих меня людях:

- Я стараюсь оставаться с людьми, которые общаются со мной, а не просто уходить в сторону, словно они меня не интересуют.
- Я хочу спорить страстно, но без злобы, чтобы меня обзывали обидными словами утром, а вечером пили со мной в знак мира и дружбы.
- Я хочу в полной мере уважать других, чтобы они были такими, какие они есть, и действовали так, как им хочется; но при этом я приложу все усилия, чтобы убедить их опробовать что-то новое.
- Я хочу делиться своими идеями, уверенный в том, что мой маленький подарок вернется ко мне стократно.
- Я хочу опробовать все возможное в области общения со своими коллегами, передачи им моих идей и получения их идей взамен.
- Я хочу уважать сильные чувства, испытываемые людьми, их верования и идеи других в такой же мере, в какой я уважаю свои.
- Я хочу проверять эти верования и идеи на прочность, сталкивая их между собой; я уверен в том, что в этих испытаниях родятся еще лучшие идеи.
- Я хочу верить, что мы делаем это из любви к делу, что мы действительно небезразличны друг другу, и что мы приветствуем горение настоящей страсти, настоящей работы, настоящего обмена идеями.

Я прикладываю все усилия, чтобы стать таким человеком. И я хочу быть рядом с другими такими же людьми. Спасибо за внимание.

Эрик Джонсон

C++ — забытый троянский конь

> Я знаю программистов, которые начали изучать C++ с единственной целью: чтобы получить возможность обозначать комментарии символами //. Несомненно, популярность C++ в значительной мере основана на простом факте: любая допустимая программа ANSI C также является программой C++, делающей примерно то же самое. — *Ред.*

Я считаю C++ интересным языком. Нет, не потому, что мой компилятор может выдать бессвязную цепочку ошибок, если я забуду включить все заголовки для умиротворения гневных богов STL. И не потому, что круг его сторонников достиг устойчивого состояния и теперь начинает медленно сужаться. C++ интересен тем, что история завоевания им сообщества программистов стала для меня хорошим уроком. Избранная им тактика была обманчиво проста, и все же многие технологи, особенно системные архитекторы, редко осваивают ее.

Чтобы понять, что произошло, необходимо запустить машину времени. До P2P, до появления спама, до появления Web, еще до того, как Интернет хотя бы отдаленно приблизился к массовым технологиям — нужно вернуться к тому времени, когда компьютеры Macintosh еще работали на классических чипах Motorola 68000. C++ родился в мире, в котором явно должен был доминировать язык C. В конце 1980-х годов C стал основным языком многих выпускников в области компьютерных технологий. Он был достаточно уважаемым для преподавания на университетском уровне и достаточно быстрым для применения в деградирующей предметной области, называемой «реальным миром». Реальная угроза владычеству C исходила разве что от таких мощных конкурентов, как Pascal, Basic, FORTRAN и Cobol. Pascal недолго грелся в лучах славы, но обгорел в них. Basic завоевал свою долю рынка, однако так и не смог стереть темного пятна со своего происхождения, сколь бы несправедливой не была такая участь. Остаются всего два реальных претендента. FORTRAN предназначался для инженеров с логарифмическими линейками, а Cobol был... Cobol, и этим все сказано. В C идеальным образом (на тот момент) сочетались уважаемый язык программирования и практичный деловой инструмент. С этих позиций он захватил рынок разработки программного обеспечения.

Конечно, я не забыл, что было немало других языков. Был язык Ada, но вся его привлекательность сводилась к 800-страничным требованиям от Министерства обороны США. Остальные потенциальные конкуренты — Modula, CLU, Smalltalk, Prolog — так и не смогли найти свою рыночную нишу, потому что они упустили потребности своей главной аудитории: студентов. Эти языки не вмещались в «персоналки», стоящие в общежитиях, не говоря уже о студенческих мозгах.

В рамках отрасли ни один язык не обосновался так глубоко, как FORTRAN, Cobol и Basic, в таком широком масштабе областей разработки. В любой отдельной системе, использовавшей эти классические языки программирования, обеспечивалось более или менее пристойное сосуществование между ними. Взаимодействие никогда не было идеальным, и все же оно несомненно было реальным. В зависимости от операционной системы было возможно обращение к Cobol из FORTRAN, и даже к FORTRAN из Basic. Была возможна совместимость на уровне компоновки.

Мощь этого тройственного союза не только прекратила все споры среди сообществ. Она означала, что дорогостоящие бизнес-консультанты, пишущие на Cobol, с определенной вероятностью могли приспособить для своих целей статистический пакет, написанный на FORTRAN. Результат всегда выглядел уродливо, но таково большинство интеграционных проектов в реальном мире. Во всяком случае, группы смогли дополнять друг друга через общее поле, которым в большинстве сред был компоновщик.

Не будем забывать, что у Mac состоялся ранний академический роман с сообществом Pascal, в результате которого родилась конвенция вызова Pascal и пристрастие к строкам Pascal. К счастью, Mac исцелился от этой глупости. Несмотря на все неудобства, C и FORTRAN продолжали взаимодействовать с Mac.

Язык программирования C очень естественно подошел для этого маленького, спокойного мирка. В большинстве систем функции C могли вызываться из FORTRAN, и наоборот. В любой конкретной системе C мог занимать доминирующее, подчиненное или равное положение со своими «коллегами» в контексте разработки.

Почему это так важно? Потому что для того, чтобы любая новая технология прижилась, она должна успешно интегрировать все существующее наследие, на месте которого она желает доминировать. Другими словами, нельзя требовать у организации, чтобы она переписывала все программы заново. Используйте то, что уже есть, и вы поможете делу.

Так что язык C пришел как помощник. Очарование C++ кроется в том, что при первом появлении он тоже представляется помощником, но при достаточном поощрении превращается в завоевателя, а в конечном счете — в нового владыку, перед которым все должны склониться. Язык C++ намеренно проектировался так, чтобы в нем вместилось как можно больше аспектов существующего языка C. Только самый наблюдательный адвокат C++ сможет выделить те области, в которых совместимость с C не сохранилась.

Что же дает эта «родовая принадлежность»?

Программисты C могут легко, почти незаметно, перейти на компилятор C++. Да, в ту раннюю эпоху существовали различия в быстродействии... да, у компиляторов C++ были свои странности. Многие программисты C принимали слишком близко к сердцу эти педантичные выходки. Непредсказуемые предупреждения

о необъявленных функциях снились им по ночам, а если и это не пронимало — от высокопарной болтовни о чудесной силе объектно-ориентированного программирования им становилось стыдно, что они пользуются этим компилятором.

Но в действительности все это означало, что программисты C могли и дальше программировать на C, просто присоединяя несколько новых букв к вызовам компилятора, и мир нисколько не изменился. Вернее, еще не изменился. Но ловушка уже была расставлена.

C++ упростил переход, потому что он легко поглотил всю функциональность в существующей области C. Ничего не надо было переписывать заново. Достаточно было внести единственное изменение: объявить все старые функции с волшебными словами «extern C», и они легко поглощались C++.

За предшествующие годы взаимодействие между функцией C и функцией FORTRAN требовало определенных усилий и понимания того, как устроены эти два мира. Как передаются вещественные значения? Производится ли передача по ссылке или по значению? Как устанавливается соответствие между логическими возвращаемыми значениями? В C++ все работало иначе. А вернее, просто работало.

Это обстоятельство дало весомые аргументы сторонникам C++, появлявшимся в организациях C. Они могли легко разрабатывать новую функциональность без потери готовой кодовой базы. Они всего лишь добавляли новый уровень в существующую систему. Вдобавок C++ позволял объявлять функции так, что даже ветераны C старой школы могли пользоваться ими. Конечно, старая гвардия C не могла получить доступ к новомодным классам и шаблонам, но это устраивало обе стороны. Преобразованные имена, вводимые компилятором C++, воспринимались как умеренное чудачество.

Поначалу C++ не создавал угрозы для ветеранов. А если организация росла, ветераны постепенно уходили в руководство, поэтому вскоре их мнение перестало быть слышным. Именно здесь наступает поворотный момент, когда C++ начинает играть мускулами. Язык не только задействует возможности существующей кодовой базы, но и предлагает своего рода попурри, «шведский стол», бесконечный источник всевозможных «примочек» и удобных возможностей, которыми может воспользоваться любой разумный разработчик.

Язык C++ никогда никому не навязывал новые возможности. Не будем путать позицию языка с позицией его ревностных последователей в сообществе. Если вы не хотели использовать новые объектно-ориентированные средства — вы их не использовали. Но они были такими соблазнительными, не правда ли? Они отливали золотым блеском, и работать с ними было очень интересно. А вы были молоды. Вы не видели ничего лучше. Вам были нужны деньги. И вообще это было интересно. Казалось, что C++ может исполнить все мечты в области программирования.

Все начинается с мелких изменений. Изменяется стиль комментариев — /* */ превращается в //. Затем объявления переменных свободно расставляются посреди функций. Просто и вроде бы незначительно, но эта возможность пришлась по сердцу многим разработчикам.

Затем начинается неизбежное. Ссылки прокладывают путь в объявления функций. Неопытный разработчик начинает возиться с перегрузкой функций, что приводит к обязательному применению корректировки имен (name mangling). Теперь

на пути других языков программирования из прошлого возникает огромная стена. Структуры превращаются в классы с виртуальными функциями. Классы обзаводятся нетривиальными конструкторами, а критические функции высокого уровня начинают генерировать исключения. Затем наступает время головоломных архитектур с множественным наследованием. В этой фазе у «традиционных» языков программирования не остается надежд на участие.

К моменту завершения метаморфозы становится слишком поздно. Возврата к прошлому нет. Красота эволюционного процесса состоит в том, что чем больше кодовая база насыщается специфическими возможностями C++, тем труднее вернуться к прошлому. Код порождает необходимость дальнейшего использования C++, а само его существование служит катализатором процесса.

С учетом трудностей, создаваемых компоновщиками в большинстве современных операционных систем, C++ приходится идти на сверхъестественные выкрутасы, чтобы его перегрузка функций была доступна для простых смертных. Но именно глупость происходящего служит лучшей защитой. Когда в игру вступают скорректированные имена, у остальных классических языков уже не остается возможности «играть на равных». Ловушка захлопнулась. Возврата к прошлому нет. C++ предпринял классический маневр: охватить, расширить, подавить. Троянский конь. C++ использовал проверенный временем метод, чтобы покорить сообщество и направить его на новую миссию.

Слишком часто мы, разработчики, полагаем, что радикальные изменения непременно сопряжены с большими, неделимыми, глобальными событиями, стоящими на развилке пути. Нам представляется, что изменения всегда сопровождаются корпоративными приказами, отраслевыми инициативами, потоками пресс-релизов и тщательно подготовленными презентациями PowerPoint. Но как и в области тектонических сдвигов и эволюции, великие и долгосрочные изменения также могут происходить постепенно в результате тщательно спланированных действий. Эффект дополнительно усиливается, если разработчики могут использовать уже существующую базу.

Эрик Липперт

Сколько работников Microsoft нужно для того, чтобы сменить лампочку?

Лично меня просто бесит, когда 16-летние гении от программирования начинают ныть о медленной работе и некомпетентности опытных программистов — в своих блогах они всегда такие храбрые и сильные, особенно когда в Интернете никто не знает о прыщах и скобках на зубах. Наверное, когда я увижу, что очередной вундеркинд на Slashdot жалуется на то, что Windows XP с ее миллионами строк программного кода «содержит больше багов[1], чем озеро в Миннеаполисе летом», и делает вид, что он (конечно, это всегда «он») — так вот, он справился бы с этим лучше, меня стошнит.

Между программированием, преподаваемым в старших классах школы, и программированием в коммерческих фирмах существует огромная разница. Чтобы написанные вами пять строк кода превратились в пять строк конечного продукта, придется сделать еще миллион шагов. Многие выпускники с дипломом специалиста по вычислительной технике не имеют ни малейшего представления о том, как проходит создание коммерческих программ за пределами написания кода.

Эрик Липперт был основным разработчиком сценарных технологий Microsoft, VBScript и JScript. Хотя после прочтения статьи начинает казаться, что Microsoft является бюрократической организацией (и в этом есть доля истины), бюрократизм возник не без причины. И такая же бюрократия потребуется любому разработчику программ, которые должны идеально работать в руках пресловутого близорукого каталонца, не создавая дефектов в системе безопасности. Благодарю Эрика за его терпение и юмор, потому что я бы приговорил этих подростков с их «всего пятью строками кода!» к 30 годам каторжных работ по написанию трехмерного текстового редактора для близоруких каталонцев на Cobol. — *Ред.*

В те времена, когда я действительно регулярно занимался добавлением новых возможностей в VBScript, мне часто присылали сообщения с просьбами реализовать те или иные функции. Чаще всего запросы были «одноразовыми» — для функ-

---

[1] Игра слов: bug — это и ошибка в программе, и насекомое. — *Примеч. ред.*

ций, решающих конкретную задачу. Скажем, «Мне нужно вызвать ChangeLightBulbWindowHandleEx, но для этого нет специального элемента ActiveX, а напрямую вызывать функции Win32 API из сценариев нельзя; нельзя ли включить метод ChangeLightBulbWindowHandleEx в список встроенных функций VBScript? Ведь это всего пять строк кода!»

Я всегда отвечаю таким людям одно и то же: если это всего пять строк кода, напишите свой объект ActiveX. Потому что вы абсолютно правы, для включения этой возможности в библиотеку времени выполнения VBScript мне потребуется примерно пять минут. Но сколько работников Microsoft в действительности понадобится для того, чтобы сменить лампочку[1]?

- Один разработчик, чтобы за пять минут реализовать ChangeLightBulbWindowHandleEx.
- Один руководитель проекта (РП), чтобы написать спецификацию.
- Один специалист по локализации, чтобы просмотреть спецификацию на предмет потенциальных проблем с локализацией.
- Один специалист по пригодности (usability), чтобы проанализировать спецификацию на предмет доступности для лиц с ограниченными возможностями и практической пригодности.
- По крайней мере один разработчик, один тестер и один РП, чтобы провести «мозговой штурм» для выявления потенциальных проблем безопасности.
- Один РП, чтобы включить модель безопасности в спецификацию.
- Один тестер, чтобы написать план тестирования.
- Один руководитель группы тестирования, чтобы обновить график тестирования.
- Один тестер, чтобы написать контрольные примеры и включить их в ночную автоматическую проверку.
- Три или четыре тестера, чтобы участвовать в выявлении ошибок именно для данного случая.
- Один технический автор, чтобы написать документацию.
- Один технический рецензент, чтобы проверить документацию.
- Один редактор, чтобы проверить документацию.
- Один руководитель отдела документации, чтобы интегрировать новую документацию в существующий текст, обновить содержание, алфавитные указатели и т. д.
- Двадцать пять переводчиков, чтобы перевести документацию и сообщения об ошибках на все языки, поддерживаемые Windows. Руководители переводчиков живут в Ирландии (европейские языки) и Японии (азиатские языки); оба места существенно сдвинуты по времени относительно

[1] Change a Lightbulb. — *Примеч. перев.*

Редмонда[1], поэтому общение с ними иногда создает непростые организационные проблемы.

- Группа старших руководителей, чтобы координировать работу всех этих людей, выписывать чеки и объяснить смысл дополнительных затрат вице-президенту.

Ни одна из этих задач по отдельности не занимает много времени, но они быстро накапливаются — и это для очень простой возможности. Обратите внимание: я исхожу из того, что все работает идеально; а если в пяти строках кода окажется ошибка? Придется прибавлять затраты на поиск ошибок, написание регрессионных тестов и т. д.

Исходные пять минут программирования оборачиваются многими человеконеделями работы и огромными затратами — и только потому, что одному человеку лень за несколько минут склепать на VB6 элемент, выполняющий нужную задачу. Простите, но в коммерческом отношении это не имеет ни малейшего смысла. Мы в Microsoft очень, очень стараемся не выпускать полусырые программы. Довести программу до ума — что среди прочего означает, чтобы близорукий испанец, говорящий на каталонском, мог легко использовать любую функцию, не опасаясь создать дефект в системе безопасности — весьма недешево! Но мы должны довести ее до ума, потому что при поставке новой версии сценарного ядра сотни миллионов людей будут использовать этот код, а десятки миллионов будут для него программировать.

Любая новая возможность, не удовлетворяющая потребностей широкого круга пользователей, фактически крадет ценные ресурсы, которые могли бы быть потрачены на реализацию функций, исправление ошибок или поиск дефектов безопасности, влияющих на жизнь миллионов людей.

---

[1] Место, где ведутся все разработки Microsoft. — *Примеч. ред.*

Майкл «Рэндс» Лопп

Что делать, когда все плохо

Пять сценариев для технических руководителей

> Я впервые услышал о «Рэндсе», когда у меня хронически не хватало времени для обновления моего собственного сайта. Некоторые читатели принялись болтать между собой о том, что бы почитать, пока я не вернусь в строй, и кто-то сказал что-то хорошее о сайте «Rands in Repose». Конечно, я туда заглянул — даже когда я слишком занят, чтобы писать, у меня всегда найдется время прочитать
>
> абсолютно
>
> каждую
>
> статью,
>
> ссылку на которую приводит этот блоггер.
>
> Я так и сделал, и вам советую то же. Закончив чтение этой превосходной статьи, отправляйтесь на http://www.randsinrepose.com и читайте подряд
>
> абсолютно
>
> каждую
>
> статью. — *Ред.*

Незадолго до моей первой презентации на WWDC[1] Кейбл Сассер из Panic[2] подарил мне футболку.

На ней было написано: «Привет, я делаю программы для Macintosh».

Пока на WWDC шла гулянка под музыку «Jimmy Eats World», я гордо расхаживал в футболке по всему городку. Коллеги быстро подметили: «Ты больше не делаешь программы... Теперь ты руководишь другими».

И это правда.

Руковожу.

Начнем по порядку.

Итак, вы стали руководителем. Поздравляю. Может, вы не блистали в программировании и решили опробовать другую стезю... а может, вас окончательно

---

[1] Конференция World Wide Developers Conference, проводимая компанией Apple. — *Примеч. ред.*

[2] Разработчик программного обеспечения для Macintosh. — *Примеч. ред.* См. http://www.panic.com/~cabel.

достали те начальники, на которых вы работали, и теперь вы хотите ПОКАЗАТЬ ВСЕМ, КАК ЭТО ДЕЛАЕТСЯ на самом деле.

Я здесь, чтобы вам помочь.

Все первые пять лет в роли начальника будут сплошными уроками. Урок № 1 начнется в тот момент, когда кто-нибудь задаст вам вопрос, и вы вдруг понимаете, что он спрашивает вовсе не потому, что вы знаете ответ — а потому, что в названии вашей должности присутствует слово «руководитель», и спрашивающий поверит любому осмысленному ответу. Кое-кто называет это авторитетом; я называю это ответственностью.

Также существуют и другие уроки. Есть большая тройка: наем, увольнение и временное освобождение. Они создадут вам массу хлопот и не дадут спать по ночам. Впрочем, если и мелочи. Вы обнаруживаете, что стали очень часто говорить «мы». Вы заметите, что стали часто повторяться... говорите одно и то же 12 разным людям. Иногда это бывает занятно, иногда — скучно, но ничто не сравнится с тем, когда наступает Кризис.

Состояние Кризиса уникально. Когда дела идут гладко, вы приходите на работу утром и спокойно попиваете чай или кофе до первого совещания в 11 утра. Кризис — совсем другое дело. К вам начинают приставать в ту секунду, когда вы выходите из лифта, и вы не успеваете проверить свою почту... до зимы.

Кризис — это умственный паралич.

Кризис — это панический страх за карьеру.

С другой стороны, преодоление кризиса дает уверенность, опыт и уважение, однако вы должны сами определить, насколько глубок кризис и как из него выйти. Если вас не интересует тема выхода из кризиса, эта статья написана не для вас. Я будут предполагать, что вы ревностно относитесь к своей профессиональной карьере. Вы хотите сделать больше. Вы хотите заработать больше денег, а если все сложится нормально — то и изменить окружающий мир.

Может, я получал недостаточно пинков от судьбы, но меня до сих пор поражают люди, плывущие по течению. Делать все по минимуму и... быть довольным? Чем именно? Возможно, кто-то не в восторге от своей работы, но мне моя нравится, поэтому давайте начнем.

1. У меня нет нужного документа, и люди на меня орут

Серьезность: малая

На ранней стадии процесса разработки продукта все только и говорят о том, как все записать. Маркетинговые спецификации, технические спецификации... спецификации, спецификации, спецификации. Часто на основе спецификаций определяются этапы работы над проектом, но обычно сроки приходят и уходят,

и никто особенно не суетится по поводу отсутствующей или неполной спецификации. И это хорошо.

На самом деле факт отсутствия конкретного документа не имеет отношения к кризису. Настоящий вопрос звучит так: «Кто требует упомянутую спецификацию и почему?»

Если требующий обладает законной потребностью в информации, вполне возможно, у вас возникли серьезные проблемы. Кто-то где-то не может выполнять свою работу, и это плохо, потому что эти люди будут злиться и «переведут стрелки» на вас.

Совет: не путайте запрос информации с требованием полного ответа. «На вопрос необходимо ответить тщательно и полностью; следовательно, я должен начать с выбора информационного шаблона, оптимальным образом соответствующего структуре ответа, и БЛА-БЛА-БЛА...» Проходит две недели, и источник запроса начинает действовать дальше. Вы упустили возможность сделать вид, будто вы знаете, о чем идет речь; не удивляйтесь, если на вас заодно повесят ярлык человека ненадежного, с которым трудно работать. Так держать.

Крупные организации действительно верят, что все необходимо документировать, и они правы. Общение в больших организациях — дело непростое, потому что у каждого есть свое мнение, и вы никогда не знаете, кому в голову придет очередная блестящая мысль. Политика крупных компаний в отношении документации усиливается многоуровневым руководством, пытающимся выяснить и измерить то, что в действительности происходит в больших группах людей (см. «Долой отчеты о состоянии дел»[1]).

Если вас беспокоит отсутствие какого-то важного документа, снова посмотрите на того, кто вызывает у вас это чувство, и спросите себя: «Что это — честный запрос фактов или «перевод стрелок»?»

В первом случае лучшим выходом всегда является общение лицом к лицу, особенно если время дорого.

При чтении этого раздела может возникнуть впечатление, что я являюсь противником документации — но это глупо, потому что Я ПИШУ БЛОГ. Вспомните контекст этой статьи: «Что делать, когда все плохо». Я говорю о ситуациях, когда все выглядит так, словно небеса обрушиваются вам на голову, и вы должны действовать быстро.

2. Важный инструмент разработки отсутствует в группе

Серьезность: зависит от инструмента

В процессе разработки программисты используют бесчисленное множество программ, но на самом деле им реально необходимы только четыре инструмента:

---

[1] См. http://www.randsinrepose.com/archives/2003/11/20/status_reports_20.html.

- Редактор
- Компилятор
- Программа контроля версии
- Программа отслеживания ошибок

Мне еще не встречалась ни одна техническая организация, в которой не было бы какого-нибудь редактора или компилятора. Но однажды меня ждал настоящий шок: в одной фирме я обнаружил отсутствие как программы контроля версии, так и программы отслеживания ошибок. Если вы когда-либо окажетесь в подобной ситуации, первым делом (еще до того, как сядете за свой стол) добейтесь, чтобы эти программы оказались на своем месте, иначе вас и вашу организацию ждет один сплошной кризис. Любую техническую организацию, в которой работает более двух людей без средств контроля версии и отслеживания ошибок, неизбежно ждет провал, как только проект начнет набирать обороты.

Эти инструменты упрощают работу программиста, позволяя всей команде:

- Взаимодействовать, не мешая друг другу. *Ты делаешь это, я делаю то.*
- Отчитываться за свою работу. *Кто проверял этот компонент? И вообще, кто допустил эту ошибку?*
- Измерять свою работу. *Сколько ошибок я допустил? А как насчет тебя?*
- Помнить, что они делали в прошлом. *Кто одобрил этот бред?*

На довольно ранней стадии вашей карьеры руководителя вы поймете, что люди заходят в ваш кабинет по одной главной причине — ради разрешения конфликтов. Когда участники конфликта перестают орать друг на друга, нужно заставить их обратиться к фактам, потому что факты вернут их на землю (а значит, у них будет меньше причин орать). Упомянутые мной программы являются отличными источниками холодных, бесстрастных фактов, и это может вам помочь.

3. Я не выношу своего руководителя программы/продукта (или его вообще нет)

Серьезность: средняя

У вас как у технического руководителя есть двое важных соратников. Первый — это руководитель продукта... из отдела маркетинга. Этот человек представляет вашу связь с клиентами и их потребностями. Второй — руководитель программы, повелевающий организацией процесса и рабочими взаимодействиями.

Роль руководителя программы определить труднее, потому что многие технические руководители путают роль руководителя программы со своей собственной ролью. Руководитель программы отвечает за весь процесс поставки конечного продукта. Представьте: вы, технический руководитель, передаете DVD с конечным продуктом руководителю программы, а он делает все необходимое для того,

чтобы продукт оказался в магазине, упакованным в красивую коробку. И не думайте, что это легко — на самом деле это огромная работа.

Руководители продукта и программы в действительности являются поставщиками информации. Руководитель продукта представляет потребности клиента — он говорит, чего хочет клиент, а вы это делаете. После завершения вашей работы он рассказывает, что было дальше. Информация руководителя программы имеет организационный характер. Он знает ответ на любой вопрос или знает, кого нужно спросить. Хорошие руководители также помешаны на формальных сторонах процесса и помнят даже о последней мелочи.

Еще несколько слов о руководителе программы: моя вера в положительную роль руководителя программы основана на печальном личном опыте. Я впервые занял должность роли технического руководителя в маленькой фирме с 20 работниками. За два года наш штат вырос до 250 человек, и я вел три линейки продуктов. К этому моменту высшее руководство решило ввести должность руководителя программ, и я немедленно попал под подозрение. «Что делают эти парни? Записывают основные моменты совещания? Господи, только время зря тратят». Неверно, все неверно. Хороший руководитель программы уделяет внимание мелочам. Он решает массу житейских проблем, связанных с выпуском программы. Вы не поверите, сколько времени он может сэкономить среднему техническому руководителю.

Если вы не можете работать с этими коллегами, или их попросту нет, начинаются проблемы. Вам придется выполнять их работу вместо них, а значит, у вас остается меньше времени на выполнение функций технического руководителя. Возможно, вы даже не почувствуете неладного из-за постоянной занятости, и все же группа и продукт будут понемногу лишаться вашего времени, пока вы будете проверять, как выглядит картинка на коробке с продуктом.

Вероятно, из этих двоих руководитель продукта (вернее, его идиотизм или отсутствие) способен создать больше проблем, потому что его данные влияют на работу всей группы. Ситуация станет еще хуже, если при нехватке времени вы заявите: «Так, я и сам знаю, что лучше для клиента». Снова неверно. Если только ваш продукт не предназначен для технических руководителей, занимающихся разработкой программного обеспечения, скорее всего, ваше мнение интереса не представляет. Извините, но это так.

4. Мой продукт не готов даже в первом приближении

Серьезность: меньше, чем вы думаете (будем надеяться)

Для начала стоит вспомнить, что группа разработки продукта сходит с ума в последнем месяце любого цикла разработки сколько-нибудь важного продукта. Правда. Начинается форменное безумие. Они слишком долго смотрели на этот чертов продукт, у них развилась сильная эмоциональная привязанность, а это приводит к иррациональному, нелепому поведению, которое не связано с реальностью. Да, это вы — мистер Обезумевший Технический Руководитель. Осталось две недели

до выхода продукта, и вы совершенно уверены, что вам в этот срок ни за что не уложиться. Вы заявляете: «Продукт никуда не годится».

Здесь возможны две равновероятные ситуации. Первая: возможно, вы стали слишком близки к продукту, чтобы осмысленно судить о нем. Близость затмевает здравый смысл, и готовность продукта с вашей точки зрения не имеет ничего общего с довольством покупателя. Если вы в момент просветления поймете, что оказались в подобной ситуации, лучше всего найти человека/сторону, суждению которого вы доверяете, и провести независимую оценку. Возможно, инстинктивное побуждение заставит вас обратиться к специалистам по контролю качества, но они тоже влюблены в продукт, и скорее всего, помешаны на качестве еще больше вас.

Может, начальник? Или другой технический руководитель? Я не знаю, кто, лишь бы это не был человек, который последние три месяца жил и дышал этим злосчастным продуктом. Когда вы найдете упомянутого здравомыслящего человека, задайте ему вопросы типа:

1. Реализована ли основная функциональность программы? Как реализована? Можно ли ее протестировать?
2. Сколько осталось ошибок?
3. Сколько ошибок вы исправляете ежедневно?
4. Какое количество ошибок можно допустить при выходе программы?
5. По каким критериям принимается решение об отсрочке исправления ошибок?
6. Какую стратегию обновления следует использовать?

Задача здравомыслящего человека — не принимать решение за вас. Он должен быть нейтральным и помочь вам сформировать решение, задавая умные вопросы. Неопытные руководители обычно не обращаются к окружающим за советом, потому что думают, что это признак слабости — и как же они ошибаются! Когда вы обращаетесь к коллегам за помощью, это позволяет им задействовать свой уникальный опыт для решения проблемы; это позволит вам принимать более качественные решения, одновременно способствуя усилению группы. Обращаться за помощью очень важно. Делайте это. Часто.

Но это была ситуация № 1... получение независимой оценки. Так мы приходим к ситуации № 2 — да, вы правы, продукт не готов даже в первом приближении, а через 14 дней его нужно сдавать. Крайний срок стремительно приближается, но почти все качают головой и бормочут: «Не готово».

Не готово. В дополнительных подтверждениях нет необходимости. Вы продолжаете лихорадочно работать, служба качества в полном шоке, а ваш руководитель программы плачет в своем офисе.

Да, продукт не готов.

Рядовые работники начинают ныть по поводу возникшей ситуации. Я имею в виду нытье в хорошем смысле слова. Это тоже способ передачи информации, и если ваш начальник прислушается, он зарегистрирует эту информацию. Проблема в том, что начальник — это вы, и теперь ВАШЕ ДЕЛО приступить к коррекции курса, потому что опоздать все же лучше чем сесть в лужу.

Если с вами это случается впервые, вам будет казаться, что ситуация хуже, чем на самом деле. Открою вам секрет: почти все считают, что технари врут, предлагая какой-то график. Это то, что любители умных словечек называют «трюизмом». Если технический специалист говорит, что на решение задачи уйдет месяц, скорее всего, ему понадобится три. Может, «врут» сказано слишком сильно. Клянусь, мы не врем, а стараемся изо всех сил. Просто мы действительно не знаем, сколько времени уйдет на выполнение работы, пока она не будет выполнена наполовину.

Организации по-разному защищаются от неточности технических оценок. В некоторых группах закладывается дополнительный допуск по времени. Другие включают в график загадочные этапы ПОСЛЕ запланированной даты поставки. Если первый срок выпуска будет сорван, вы будете приятно удивлены, когда руководитель продукта спросит: «Сколько еще времени вам понадобится?» Что, вы и не знали об этом? Теперь знаете.

ПРЕДУПРЕЖДЕНИЕ: если вы хотите, чтобы ваши слова пользовались доверием в организации, учтите, что нарушение сроков на поздней стадии разработки повредит вашей репутации. Любая нормальная команда из технического руководителя и руководителя программы определяет контрольные точки графика для сверки и внесения поправок, чтобы итоговое расписание стало более надежным. Изменения графика в последнюю минуту нарушают третье правило управления проектами Рэндса: «Никаких сюрпризов».

5. Меня тошнит от моей компании/работы

Серьезность: высокая

Истинная история. В начале 1990-х годов компания Borland получила тяжелый удар от Microsoft. Перевод офисных приложений Borland на платформу Windows проходил ужасно. Монополия Microsoft действовала в полную силу... вышла первая версия пакета Office, которая подкосила конкуренцию своей низкой ценой. Прощайте, Quattro Pro, Paradox и dBASE.

После нескольких лет расширения и переезда в потрясающий (до сих пор) кампус, компания Borland стояла на грани развала, и я это сознавал. Выявление проблемы становится первым шагом к преодолению кризиса в подобных ситуациях... вы знаете, что корабль тонет, хотя высшее руководство продолжает излучать оптимизм на всех общих собраниях. Его оптимизм понятен; если рядовые работники поверят, что конец близок, народ попросту начнет разбегаться.

Что я сделал в этой ситуации? Покинул тонущий корабль. Я припомнил свою техническую квалификацию и перешел в компанию, занимающуюся разработкой баз данных (ныне не существующую). Проблема была в том, что новая компания находилась в куда худшем состоянии, чем Borland. Я не знал, что руководитель службы персонала, нарисовавший радужную картину, уволился через месяц после разговора со мной. Внезапно мне пришлось отлаживать системы сборки и пить очень плохой кофе в компании разработчиков баз данных, страдающих хронической депрессией. Такие дела.

Возможно, это покажется очевидным, но если у вашей компании дела идут плохо, выход из кризиса состоит из двух частей. Первая — выявление проблемы. В Borland есть люди, которые и до сегодняшнего дня продолжают ныть по поводу компании точно так же, как они это делали десять лет назад. Тем не менее, несмотря на все свое нытье, они никогда ничего не делают для исправления положения.

Вы не относитесь к их числу, потому что вы все еще продолжаете читать. Вы хотите что-то сделать с возникшими проблемами. Вы хотите двигаться наверх. Вы хотите попасть туда, где вы:

А) получите прибавку;

Б) получите повышение;

В) будете заниматься тем, что вам интересно;

Г) будете работать на компанию, от которой вас не тошнит.

Уходя из Borland, я успешно решил задачи А и Б, но не справился с Г (и как выяснилось позднее, и с В). Это был худший переход в моей карьере. Мне понадобился почти год, чтобы получить место, где я действительно начал двигаться вперед. Хорошие руководители поддерживают свои группы, свои продукты и свою карьеру в хорошем темпе.

Темп.

Этот термин лучше, чем привычная «мобильность», ориентированная снизу вверх. Темп как постоянное движение вперед.

Как вы собираетесь обеспечить свой личный темп — это ваше личное дело. Мой поток советов из области руководства, который может показаться бесконечным, в действительности представляет собой именно это... советы. В действительности вам бы пригодился мой личный опыт, но я не могу им поделиться, потому что получить его можно только одним способом — вы должны сами оказаться в кризисной ситуации. Когда вы окажетесь по колено в неприятностях и будете основательно перепуганы, возможно, какой-нибудь из моих советов придет вам в голову — а может быть, вы придумаете свое, более элегантное решение. Как бы то ни было, после выхода из кризиса вы начнете двигаться быстрее... а может, медленнее.

Ларри Остерман

равила разработки программного обеспечения от Ларри: не оценивайте труд тестеров по тестовым метрикам

Ларри Остерман начинает свою статью словами: «Вероятно, эта статья покажется немного спорной». Я с ним не согласен. Мне эта статья совершенно не кажется спорной: она абсолютно и полностью правдива относительно того, что любые попытки измерить качество или производительность тестеров по тестовым метрикам обречены на неудачу. Просто большинство людей об этом не знает. Пока.

В сущности, этот принцип также относится к разработчикам, руководителям и вообще почти всем, кто занимается интеллектуальной работой. В тот момент, когда вы попытаетесь измерить качество тестера или разработчика по любой метрике, которая только придет вам в голову, тестеру или разработчику потребуется всего несколько минут, чтобы оптимизировать свою работу по этой метрике — то есть обхитрить систему для получения высоких показателей с одновременным сокращением их истинной производительности.

Нам часто кажется, что метрики можно слегка подстроить, чтобы заставить людей работать лучше. Опыт показывает, что это не так. Подгонка метрик приводит к подгонке итоговых показателей, но реально вы ничего не добиваетесь. Не работает сама базовая идея измерения.

В своих продуктах я хорошо сознаю справедливость этого утверждения. Спроектированная мной программа отслеживания ошибок FogBugz не включает метрики производительности и не дает возможности легко определять их. В своих маркетинговых материалах мы указываем, что при любых попытках наказывать программистов за ошибки в коде можно быть уверенным лишь в одном: что рано или поздно количество «ошибок» в базе данных уменьшится до нуля, тогда как количество ошибок в программе останется прежним. Мы не хотим, чтобы система FogBugz превратилась в дубину для службы персонала. Всего неделю назад я общался по телефону с клиентом, которому нужна была любая система отслеживания ошибок, позволяющая собирать статистику относительно того, «сколько времени у программистов уходит на исправление ошибок». Я объяснил, почему в нашей системе эта возможность отсутствует, и скорее всего, никогда не появится в будущем, даже если из-за этого мы потеряем клиента. Я предпочитаю потерять клиента, чем видеть, как нашу программу обвиняют в чужих управленческих ошибках. И знаете что? Ничего

страшного не произойдет. Руководители все равно знают своих лучших тестеров и разработчиков. Это те люди, о которых говорит Пол Грэхем в своей статье «Великие хакеры». Чтобы понять, кто не справляется со своей долей работы, вполне достаточно обычных мыслительных способностей, заложенных в ваш мозг при рождении. Никакие искусственные метрики, которые так легко обходятся, для этого не нужны. — *Ред.*

Вероятно, эта статья покажется немного спорной ☺.

Среди руководителей групп тестирования встречается нездоровая тенденция измерять производительность своих подчиненных по количеству ошибок, о которых они сообщают. Насколько мне удалось разобраться, их логика выглядит примерно так.

РУКОВОДИТЕЛЬ 1. Так, нам нужны конкретные метрики для оценки производительности работы наших тестеров. Какие будут предложения?

РУКОВОДИТЕЛЬ 2. Лучший тестер — тот, кто находит больше всего ошибок, верно?

РУКОВОДИТЕЛЬ 1. Логично. Давайте оценивать их работу по количеству представленных ими ошибок!

РУКОВОДИТЕЛЬ 2. Хмм. Но ведь тестеры могут обойти систему — они будут сообщать о десятках выдуманных ошибок, чтобы накрутить свои счетчики...

РУКОВОДИТЕЛЬ 1. Верно. Как бы это предотвратить? ... Я знаю, давайте оценивать их по количеству ошибок, помеченных как «исправленные» — ошибки с пометками «не поддается исправлению», «обусловлено архитектурой» или «не удается воспроизвести», не будут учитываться в метрике.

РУКОВОДИТЕЛЬ 2. Вроде бы должно сработать; сейчас я разошлю сообщения среди наших тестеров.

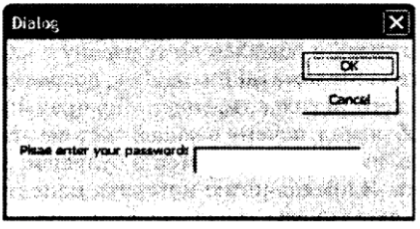
Звучит неплохо, не правда ли? Работа тестеров будет оцениваться по абсолютной величине, вычисленной по количеству найденных ими реальных ошибок — не фиктивных, а настоящих ошибок, требующих внесения изменений в продукт.

Проблема в том, что на практике эта идея не работает.

Тестеры получают мощный стимул для мелких придирок — вместо того, чтобы искать значительные ошибки в продукте, они пытаются выявлять ошибки для увеличения своих счетчиков. И это создает конфликты с разработчиками, если последние посмеют отнести найденные ошибки к любой категории, кроме «исправлено».

Давайте посмотрим, как это происходит, на следующем простом примере.

В моем приложении вызывается диалоговое окно следующего вида:



Без введения специальных метрик большинство тестеров зафиксируют «Множественные ошибки в диалоговом окне ввода пароля»; под этим понятием подразумеваются орфографическая ошибка и неточное выравнивание подписи к тестовому полю.

Возможно, они также зафиксируют отдельную ошибку локализации, так как подпись отделена от текстового поля слишком малым промежутком (отдельную, потому что эта ошибка относится к другой категории).

Но если производительность труда тестера будет оцениваться по количеству найденных им ошибок, у него появляется стимул сообщать о как можно большем количестве ошибок. Так одна ошибка превращается в две — опечатка и неверное выравнивание.

В этом варианте проблема абсолютно ничтожна; для меня исправить одну ошибку ничуть не сложнее, чем две.

Но что произойдет, если проблема не является настоящей ошибкой — вспомните, ошибки типов «не поддается исправлению» или «обусловлено архитектурой», не включаются в метрику, чтобы тестеры не заполняли базу данных вымышленными ошибками с целью искусственной накрутки своих счетчиков.

ТЕСТЕР. Если создать файл, войдя в систему в качестве администратора, то поле владельца в дескрипторе безопасности файла устанавливается равным BUILTIN\Administrators вместо текущего пользователя.

Я. Да, так и должно быть, поэтому я отношу ошибку к категории «обусловлено архитектурой». Дело в том, что Microsoft® Windows NT® считает всех администраторов взаимозаменяемыми, поэтому при создании файла участником BUILTIN\Administrators в качестве владельца указывается группа, чтобы любой администратор мог изменить DACL для файла.

Обычно дискуссия на этом заканчивается. Но если производительность тестера оценивается по количеству выявленных им ошибок, у него появляется стимул оспаривать любую классификацию ошибок, кроме «исправленных». Поэтому разговор продолжается:

ТЕСТЕР. Архитектура тут ни при чем. Покажи, где в спецификации указано, что в качестве владельца файла должна быть указана учетная запись BUILTIN\Administrators.

Я. В спецификации этого нет. Так работает NT; это особенность системы.

ТЕСТЕР. Тогда я списываю ошибку на твою спецификацию, потому что в ней этот момент не документирован.

Я. Погоди — моя спецификация не должна объяснять все тонкости инфраструктуры безопасности операционной системы. Если у тебя возникли проблемы, обращайся к авторам документации NT.

ТЕСТЕР. Нет, это ТВОЯ проблема — спецификация написана плохо, так что исправляй ее. Я приму «архитектурную» классификацию только в том случае, если ты мне покажешь спецификацию NT, в которой описано это поведение.

Я. (вздыхая). Ладно, записывай ошибку на спецификацию, а я посмотрю, что можно сделать.

Дальше возможны два пути: либо я должен документировать все тонкости внутреннего поведения (а в области безопасности таких тонкостей очень много, особенно в отношении наследования ACL), либо я должен преследовать разработчиков NT, сообщать о неточности и заставлять вносить ее в спецификации. Ни один из этих путей не приблизит меня к сдаче продукта. Возможно, это будет способствовать улучшению документации NT, но к МОЕЙ задаче это не относится.

Кроме того, выясняется, что показатель «наибольшего количества выявленных ошибок» изначально ущербен. Тестеры, регистрирующие наибольшее количество ошибок, не всегда обеспечивают наилучшую проверку продукта. Часто наибольшую пользу для группы приносит тот тестер, который готов потратить дополнительное время на анализ внутренних причин, и вместе с перечнем ошибок приводит подробную информацию об их возможных причинах. Но такие тестеры не могут похвастаться наибольшей плодовитостью, потому что им приходится расходовать дополнительное время на проверку четкой воспроизводимости ошибки и сбор информации о происходящем. Время, которое можно было бы потратить на поиск всяких мелочей, тратится на проверку «качественности» ошибок — иначе говоря, ошибок, способных помешать выпуску окончательной версии продукта, а не придирок типа «подпись смещена на полмиллиметра».

Я не хочу сказать, что метрики — это плохо. Вовсе нет. Однако оценка производительности людей на основе этих метрик — верный путь к катастрофе. В команде Windows NT 3.1 был один тестер, помешанный на проблемах целостности данных, и неустанно выискивавший эти ошибки. Правда, он нашел их не очень много. Зато каждая обнаруженная ошибка была абсолютно критична для успеха продукта. Если бы его работа оценивалась по метрике «количества найденных ошибок», результат был бы просто жутким, потому что его показатели были относительно низкими по сравнению с другими тестерами в этой организации. Однако его ценность для организации была ничуть не меньшей, а то и большей, чем ценность любого другого тестера. Тем не менее, абсолютная метрика не признала бы его вклад в создание продукта.

Небольшое дополнение: уже после написания исходной версии этой статьи я обсудил ее с парой разработчиков за обедом. По словам одного из моих собеседников, я забыл упомянуть о том, что оценка производительности должна складываться из двух этапов:

- Измерение: дайте мне число, представляющее качество работы, выполняемой данным пользователем.
- Оценка: насколько хорошо справляется работник со своим делом с учетом этой оценки? Иначе говоря, метрике требуется присвоить определенный весовой коэффициент — насколько она важна для общей производительности?

Далее мы перешли к обсуждению того факта, что любая метрика абсолютно бесполезна, если только она не подвергается постоянной переоценке для определения ее релевантности — метрика полезна лишь в той степени, в какой она действительна.

Также мои собеседники отметили, что абсолютное количество ошибок не может использоваться для оценки тестера. Тестер, который потратил две недели

на отыскание четырех ошибок переполнения буфера, скорее всего, принесет больше пользы, чем тестер, потративший те же две недели на выявление 20 тривиальных ошибок. В качестве метрики было предложено использовать значение поля «серьезности» ошибки, но мой коллега отметил, что это имеет смысл только в том случае, если значение этого поля имеет реальный смысл, а это часто не так (объективно определить относительную серьезность ошибки может быть очень трудно, и заполнение этого поля часто поручается самому тестеру — такая стратегия открывает широкие возможности для злоупотреблений, если только все ошибки не проходят внешнюю классификацию, а этого часто не происходит).

К концу дискуссии мы согласились на том, что количество ошибок может быть интересной, но никак не единственной метрикой.

Мэри Поппендик

Компенсация для групп

Величайший миф в области управления кадрами — утверждения о возможности создания «детсадовской» модели экономики, в которой работники, которые справляются со своей работой, награждаются, а плохие — наказываются. Для этого к личным графикам хороших работников приклеиваются маленькие золотые звездочки, и им дают достаточно премиальных, чтобы они могли купить себе булочку или леденец, и тем самым добиться еще лучших результатов.

Полная чушь.

Это настолько неверно, что от одной мысли я начинаю бормотать что-то невнятное и сердито плеваться — что мешает мне четко объяснить, почему же это неверно. И все же все известные мне крупные фирмы-разработчики и большинство мелких фирм по-прежнему пребывает в системе критериев определения качества, взятой из XIX века, хотя им уже тщательно и досконально объяснили, что эта система неверна, неверна, неверна!

Пффф.

К счастью, Мэри Поппендик достаточно хладнокровна. Она спокойно и терпеливо объясняет нам (в смысле — вам), почему критерии определения качества для специалистов умственного труда всегда контрпродуктивны. Причем она делает это с гораздо меньшим количеством — как бы это выразиться? — презрения и желчи, чем это сделал бы я... даже если бы мне удалось достаточно успокоиться, чтобы написать на эту тему. — *Ред.*

Группа прекрасно справилась со своей работой, и она это знала. Цикл за циклом она приближалась к созданию нового программного продукта, а когда подошел срок сдачи, все, что должно было работать, действовало просто идеально. На дневном совещании вице-президент похвалил всех, кто внес свой вклад в работу, а участники группы поздравили друг друга, одновременно припоминая самые мучительные моменты за последние полгода.

На следующее утро

На следующий день Сью, руководитель проекта, отвечала на давно заброшенную электронную почту, когда ей позвонил Дейв, старший по разработкам. «Знаешь,

Сью, — сказал он, — твоя команда отлично потрудилась! Я ждал запуска продукта, чтобы не беспокоить тебя раньше, но на следующей неделе подходит крайний срок. Мне нужна твоя оценка каждого участника группы. И если можно, постарайся упорядочить участников по степени вклада в работу, от наибольшего к наименьшему».

У Сью весь мир померк перед глазами. «Я не могу этого сделать, — сказала она. — Все трудились на 100 процентов. Иначе у нас бы ничего не вышло. Более того, сотрудничество является центральным стержнем наших гибких методологий».

«Но, Сью, — сказал Дейв, — должен же быть в команде самый ценный участник, за ним кто-то второй, и так далее».

«Вообще-то нет, — ответила Сью. — Но я могу оценить вклад каждого в проект».

Сью заполнила оценочные анкеты на каждого участника группы. Она попыталась оценить производительность каждого, но выяснилось, что для каждого участника ей пришлось поставить галочку в поле «Значительно превосходит ожидания». Ведь своевременная сдача продукта была впечатляющим достижением, которое значительно превосходило всеобщие ожидания.

Последствия

Спустя два дня Сью позвонили. Это была Дженис из службы персонала. «Сью, — сказала она, — твоя команда отлично потрудилась! И спасибо за то, что ты заполнила оценочные анкеты. Только знаешь, нельзя ставить всем высшую оценку. Средний показатель должен быть "Удовлетворяет ожиданиям". "Значительно превосходит ожидания" может быть только у одного или двух человек. И кстати говоря, раз ты не упорядочила участников группы, ты бы не могла зайти на наше совещание на следующей неделе? Нам понадобятся твои данные. В конце концов, в этой компании платят за эффективность, поэтому мы должны тщательно оценить всех, чтобы никто не мог усомниться в нашей добросовестности».

У Сью снова земля ушла из-под ног. В прошлом, когда у нее возникала особенно сложная проблема, она всегда консультировалась с группой, и группа всегда предлагала творческие решения; Сью решила снова поступить так же. Она подумала, что ей удастся убедить группу выбрать одну-две «звезды первой величины», и это поможет ей внести некоторое разнообразие в оценки.

На следующее утро вся группа слушала, как Сью объясняет возникшую проблему. Однако ее ждало разочарование и удивление — вместо того, чтобы с энтузиазмом взяться за решение проблемы, участники группы «сдулись» так же быстро, как и она сама. В лучшем случае они настаивали, что все работали на 200 процентов, что все помогали друг другу, и что они считали, что каждый участник замечательно справился с порученной работой. Народ явно не собирался выбирать «самого полезного», но зато все согласились выбрать самого бесполезного: безымянного начальника, который заставляет Сью выбирать среди них.

И тогда у Сью действительно возникли проблемы. Она понятия не имела, что ответить Дейву и Дженис, а ее план по привлечению группы лишь породил гнев

и подозрения. Завтра они должны приступить к совместной работе над следующей версией. Как то, что должно было способствовать повышению эффективности, так основательно сокрушило дух коллективизма?

Сью была не единственной, у кого возникли проблемы с оценкой деловых качеств и деловыми рейтингами. Один из величайших мыслителей XX века, У. Эдвардс Деминг, написал, что рейтинговые системы, премиальные и поощрительные надбавки причинили неизмеримый вред.

Деминг считал, что каждое предприятие является системой, и производительность труда отдельных работников определяется в основном режимом работы системы. В его представлении, система создает 80 процентов проблем в бизнесе и ответственность за работу системы несет руководство. Он писал, что никакие увещевания и премиальные, заставляющие рядовых работников решать проблемы руководства, попросту не работают. Деминг возражал против рейтингов, потому что они уничтожают гордость за квалификацию, и против поощрительных надбавок, потому что они лечат симптомы проблем вместо их глубинных причин.

Воспринимать доводы Деминга напрямую нелегко; ведь компании используют системы материального поощрения в течение многих лет, и область их применения только растет. Более того, Деминг в основном участвовал в промышленном производстве, поэтому вполне возможно, что его выводы не распространяются напрямую на такие области, как разработка программного обеспечения. И все же мнение такого умного человека, как Деминг, нельзя игнорировать; давайте повнимательнее присмотримся к процессу оценки работников и к системам премирования, и выясним, что же нарушает их работу.

Нарушение № 1: Конкуренция

Как инстинктивно осознала группа Сью, оценка людей для материального поощрения сталкивает работников друг с другом и мешает сотрудничеству — краеугольному камню гибких методологий. Даже когда оценки не предаются гласности, сам факт их составления не держится в секрете. Иногда результаты оценки используются как основание для увольнения работников из нижней части списка, из-за чего эта практика выглядит еще более угрожающей. Когда участники группы начинают конкурировать друг с другом за место под солнцем, дух коллективизма быстро испаряется.

Конкуренция между группами (вместо отдельных личностей) может показаться хорошей идеей, но она может быть столь же опасной. Однажды я работала в организации, в которой две разные команды разработчиков производили программные продукты, рассчитанные на один рыночный сектор. Участники команды, завоевавшей больший сектор рынка, с большей вероятностью получали надежную работу и хорошие возможности для карьерного роста. Таким образом, каждая команда расширяла способности своего продукта для привлечения большего числа покупателей. Все это привело к тому, что команды начали яростно конкурировать между собой за одну клиентскую базу и за ресурсы организации. В конечном итоге оба продукта провалились. У одного продукта шансов на успех было бы больше.

Нарушение № 2: Чувство несправедливости

Ничто так не вредит системе поощрений, как ощущение ее несправедливости. При этом не так уж важно, справедлива система или нет. Если она воспринимается людьми как несправедливая, то те, кто думает, что их незаслуженно обошли, быстро утрачивают стимулы к работе.

Чувство несправедливости возникает тогда, когда люди не получают премий, на их взгляд — заслуженных. Что, если бы вице-президент выдал Сью большую премию, но не стал бы премировать группу? Даже если бы Сью признала усердную работу своих коллег, скорее всего, они бы посчитали, что она наживается за их счет. Можете не сомневаться: после этого Сью было бы нелегко вызвать энтузиазм для работы над следующей версией, даже если бы трудная ситуация с оценками осталась бы незамеченной.

А вот другой сценарий: что произошло бы, если бы группу Сью пригласили на обед с вице-президентом, и каждому участнику вручили приличную премию? На следующий день обслуживающий персонал, работавший вечерами и по выходным, чтобы выпустить продукт вовремя, узнает об этом и почувствует себя обделенным. Разработчики, которые взяли на себя второстепенные задачи, чтобы их коллеги могли посвятить все время работе над продуктом, тоже решат, что их незаслуженно обошли. Другим группам покажется, что они справились бы с этой работой ничуть не хуже, но их поставили на «невыгодный» продукт.

Нарушение № 3: Чувство невозможности

Группа Сью уложилась в срок благодаря одному из принципов гибких методологий — выпуску высококачественного продукта, содержащего только приоритетные функции. Но давайте рассмотрим другой сценарий: предположим, группа получила список функций, который должен быть готов к определенному сроку; и список, и срок не подлежат обсуждению. Далее предположим, что группа на 100 процентов уверена в том, что выполнить задачу в срок невозможно (напоминаю, что ситуация чисто гипотетическая; конечно, в реальной жизни такое невозможно). Наконец, давайте предположим, что в случае соблюдения срока участникам группы обещана большая премия. В этом сценарии возможны два исхода. Материальный стимул работает весьма мощно, поэтому возможно, команда найдет способ сделать невозможное. Но более вероятен другой вариант: обещание недостижимой премии вызовет у группы приступ цинизма, и у нее останется еще меньше мотивов для соблюдения срока, чем до предложения премии. Когда люди чувствуют, что руководство просто уговаривает их сделать явно невозможное, вместо того, чтобы помочь и сделать задачу более реальной, предложенная премия их обычно оскорбляет, и они сдаются, даже не пытаясь ничего сделать.

Нарушение № 4: Субоптимизация

Недавно я слышала о владельце одного предприятия, который платил тестерам по 5 долларов за каждый дефект, выявленный в продукте, готовом к выпуску

бета-версии. Он думал, что тем самым стимулирует усердие тестеров, но результат получился совершенно иным. Хорошие рабочие отношения между разработчиками и тестерами ухудшились — ведь тестеры не были заинтересованы в том, чтобы помогать разработчикам быстро находить ошибки и исправлять их раньше, чем они начнут размножаться. В конце концов, чем больше ошибок найдет тестер, тем больше денег он заработает.

Оптимизация части цепочки неизбежно приводит к субоптимизации общей производительности. Одним из самых очевидных примеров субоптимизации можно считать отделение разработки программного обеспечения от поддержки и сопровождения. Если премировать разработчиков за соблюдение графика, даже если они поставляют ненадежный код без комплексов автоматизированных тестов или полноценной программы установки, дело добром не кончится. Поддержка и сопровождение системы обойдутся гораздо дороже, чем было сэкономлено при разработке.

Нарушение № 5: Уничтожение внутренней мотивации

Среди родителей бытуют два подхода к карманным деньгам у детей. Теория А гласит, что дети должны зарабатывать свои карманные деньги; деньги обмениваются на работу. Теория Б гласит, что дети должны бесплатно вносить свой вклад в домашнее хозяйство, а карманные деньги не считаются оплатой работ по дому. Я знаю одного отца, который воспитывался на теории Б, но применительно к своим детям перешел на теорию А. Он оценил каждую домашнюю работу и еженедельно платил своим детям за выполненные дела. В течение какого-то времени схема работала нормально, но потом дети поняли, что у них есть выбор, и стали избегать тех работ, которые им не нравились. А когда дети повзрослели и стали самостоятельно зарабатывать, они вообще перестали делать что-либо по дому, и отцу пришлось самому косить траву на лужайке вместе с соседскими детьми. Отец говорит, что если бы он мог начать все заново, то не стал бы привязывать карманные деньги к конкретной работе.

Примерно также дело обстоит и с работой: когда работники привыкают получать финансовое вознаграждение за достижение поставленных целей, они начинают работать только ради премий, но не ради внутреннего удовлетворения от успешно выполненной работы и своего вклада в процветание компании. Многочисленные исследования показали, что внешняя мотивация (звания и деньги) со временем уничтожают внутреннюю мотивацию, обусловленную самой работой.

Спустя неделю

Входя в комнату, где проводилось собрание, Сью изрядно нервничала. Она обсудила проблему со своим начальником Уэйном; хотя у него не было простого решения. Уэйн предложил представить проблему перед руководством. Вскоре после начала заседания Дженис спросила Сью, как она оценивает участников своей команды. Сью сделала глубокий вдох, уловила ободряющую улыбку Уэйна и объяснила, что вся идея с оценками не имеет смысла для работы в группах, особенно при

использовании гибких методологий. Она рассказала, как решила посоветоваться с группой, и как это вызвало гнев и подозрения.

«Тебе не следовало обсуждать эту тему с группой», — сказала Дженис.

«Постой-ка, — перебил Уэйн. — Я думал, что наша цель в этой компании — быть честными. Как можно держать в секрете политику оценки и ожидать, что люди будут считать ее справедливой? Неважно, если мы думаем, что она справедлива; важно, чтобы ее считали справедливой те, кто у нас работает. Нельзя принимать решения за закрытыми дверями, а потом говорить: "Не беспокойтесь, все справедливо"».

Сью поразило, насколько изменился характер дискуссии после того, как Уэйн пришел ей на помощь. Очевидно, она была не единственной, кто считал затею с оценками неудачной. Все согласились с тем, что группа Сью отлично поработала, и новый продукт должен был стать ключевым для их бизнеса. Никто не думал, что группа справится с поставленной задачей; в самом деле, успех группы в целом превзошел все ожидания. Стало очевидно, что в комнате не было ни одного человека, желавшего узнать, кто работал больше, а кто меньше, поэтому высшие оценки Сью для всех членов группы были приняты. Что еще важнее, собравшиеся поняли, что демотивация команды могла стать серьезной проблемой.

В конечном счете вице-президент согласился посетить следующее собрание группы и обсудить политику оценки персонала. Сью была уверена, что это в значительной мере восстановит командный дух.

Теперь у руководства возникла своя проблема. Было понятно, что система материального поощрения должна была сохраниться, но возникло подозрение, что способ ее реализации необходимо продумать заново. Поскольку подобные изменения не происходят за один день, был сформирован комитет по рассмотрению различных систем оценки и материального поощрения.

Для начала комитет согласился с тем, что системы не должны удивлять работников неожиданной оценкой производительности их работы. Циклы обратной связи по эффективности не должны быть ежегодными и даже ежеквартальными. Оценка становится хорошим моментом для анализа и обновления планов развития работника, но если это единственный момент, когда человек узнает об успешности своей работы — значит, менять нужно не систему оценки, а гораздо больше.

С учетом сказанного, комитет разработал ряд рекомендаций по различным формам дифференцированной оплаты.

Рекомендация № 1: Проверьте систему повышения по службе

Во многих организациях существенная прибавка к жалованию обусловлена не премиями, а повышениями с переходом в более высокооплачиваемую категорию. Если повышения недоступны, как это часто бывает у преподавателей, системы материального поощрения отчасти теряют эффективность, потому что заработать побольше можно только в случае повышения выплат. При возможности повышения работники обычно игнорируют систему выплат за заслуги и сосредоточивают внимание на системе повышения по службе. Конечно, система повышений

подталкивает к переходу в руководящий состав по мере исчерпания возможностей карьерного роста в технической области. Компании решают эту проблему при помощи «двойных лестниц», обеспечивающих выплаты уровня руководящих работников для технических гуру.

Основанием любой системы повышения является серия должностных категорий, для каждой из которых определяется диапазон зарплат, соответствующий стандартам отрасли и средним региональным показателям. Работники должны правильно распределяться по категориям, чтобы их навыки и ответственность соответствовали требованиям для их уровня. Начальное распределение и решения о повышении должны быть тщательно продуманы и одобрены руководством.

Обычно категории включаются в название должностей, а при повышении переход в новую категорию обозначается через присваивание новой должности. Как правило, должностная категория считается общедоступной информацией. Если работники достаточно справедливо распределены по категориям, а повышение происходит только при четком выполнении обязанностей в новой категории, различия в уровне оплаты на основании должностной категории обычно воспринимаются людьми как справедливые. Таким образом, группа может содержать как старших, так и младших сотрудников, специалистов общего профиля и узких специалистов, зарабатывающих разные суммы. Пока система определения должностных категорий и повышений остается прозрачной и воспринимается как справедливая, с системами дифференцированной оплаты такого рода проблем обычно не бывает.

Руководство в компании Сью решило сосредоточить усилия на процессе повышения, который бы не зависел от системы оценок или квот. Вместо этого для каждого уровня устанавливались четкие критерии повышения; когда кто-то удовлетворял этому критерию, он считался достойным повышения. Комитет рассматривал каждое предложение о повышении и достигал консенсуса по поводу выполнения критериев. Механизм его работы мало чем отличался от существующих комитетов, рассматривающих заявки на заполнение вакантных руководящих должностей.

Рекомендация № 2: Отведите на второй план систему выплат по заслугам

Если основным инструментом существенного повышения жалования является повышение, очень важно уделять как можно больше внимания справедливости системы повышения. Когда речь заходит о системе оценок, определяющих выплаты по заслугам, не стоит делать упор на сортировке людей. Исследования показывают, что при необходимости обмена информацией и координации совместной работы организации, в которых различия в уровне максимальной и минимальной оплаты сокращены, лучше работают в долгосрочной перспективе.

Используйте оценки главным образом для того, чтобы работники находились на подходящем уровне в своих должностных категориях. Оценка поможет выявить тех, кто готов к повышению, и тех, кто нуждается в дополнительном внимании, но она должна инициировать отдельный процесс повышения или исправительных действий. Достаточно около четырех оценочных градаций, и компетентный

начальник с хорошими критериями оценки и информацией из соответствующих источников сможет обеспечить справедливые оценки для достижения этих целей.

Даже когда ежегодные повышения не связаны с заслугами напрямую, оценки все равно будут много значить для работников, поэтому следует особо постараться сделать их честными и сбалансированными. За последнее десятилетие для управления оценками приобрели популярность карты сбалансированных показателей — по крайней мере, в теории. Карты сбалансированных показателей гарантируют, что внимание будет уделено всем многочисленным аспектам работы руководителя. Простая версия карты также может использоваться для оценок, предназначенных для определения выплат по заслугам; в сущности, они подчеркивают тот факт, что эффективность подразумевает хорошую работу по многим направлениям. Начальник может разработать для каждого работника карту показателей, учитывающую результаты группы, новые знания, качества лидера и т. д. Важно, чтобы работники считали исходные данные карт объективными и честно представляющими различные аспекты их работы. Постарайтесь сохранить систему простой, потому что излишняя сложность привлечет ненужное внимание к системе оплаты. Наконец, карты показателей не должны служить основанием для рейтинг-системы.

Рекомендация № 3: Свяжите участие в прибылях с экономическими факторами

Компания Nucor Steel решила войти в отрасль производства стали в 1968 году, и 30 лет спустя стала крупнейшей сталелитейной компанией в Соединенных Штатах. Когда Nucor Steel делала свои первые шаги, компания Bethlehem Steel считала ее обычной мошкой, но через 35 лет Bethlehem Steel не только обанкротилась, но и распродала свои активы. Таким образом, Nucor Steel — очень успешная компания, которая правильно вела себя в непростой отрасли. Как ни удивительно, в Nucor существует многолетняя традиция оплаты за производительность. Как компания избегает нарушений, связанных со стимулированием?

Компания Nucor Steel началась с осознания того факта, что ее ключевым экономическим фактором была прибыль на тонну готовой стали, и ее план участия в прибылях базировался на вкладе, вносимом группой для улучшения этого показателя. Например, группа, успешно разработавшая новую технологию изготовления стали или запустившая новый завод по графику, не увидит возрастания оплаты до тех пор, пока технология или завод не повлияет на прибыль компании на тонну стали. Таким образом, Nucor избегает субоптимизации за счет как можно более тесной привязки дифференцированной системы оплаты к основному экономическому фактору своей отрасли.

Рекомендация № 4: Награждайте на основании сферы влияния, а не сферы контроля

Традиционно считается, что людей следует оценивать по результатам, находящимся под их контролем. Тем не менее, оценка индивидуальных результатов вместо

групповых создает конкуренцию вместо того, чтобы способствовать сотрудничеству между членами групп. Для поощрения сотрудничества Nucor следит за тем, чтобы формула участия в прибылях награждала относительно крупные команды, а не отдельных личностей или малые группы, напрямую отвечающие за данную область. В соответствии с их принципами и политикой, если программный продукт приводит к заметному возрастанию прибыли, то награда разделяется между всеми, включая человека, предложившего идею, разработчиков и тестеров, вспомогательный персонал и конечных пользователей. Такая система материального поощрения особенно хорошо укладывается в среде гибких методологий, которая естественным образом распространяет принцип всеобщего участия (конечные пользователи, тестеры, поддержка и т. д.) на процесс разработки.

Nucor Steel усердно работает над созданием системы обучения: эксперты переезжают с одного завода на другой, машинисты играют заметную роль в выборе и запуске новых технологий, а практические знания быстро распространяются в компании. Система награждения поощряет обмен знаниями; для этого люди награждаются за их влияние на успех в областях, не находящихся под их прямым контролем.

Как именно награды могут основываться на сфере влияния, а не на сфере контроля? Я рекомендую применять методику, называемую «повышающей оценкой». Как бы вы ни старались оценивать высококвалифицированную работу, какой бы хорошей ни была созданная вами карта показателей, все равно что-то останется неизмеренным. Со временем потерянная область будет упущена из виду, и начнутся проблемы. Существует тенденция включения дополнительных показателей в карты для привлечения внимания к упущенным областям.

Тем не менее, у проблемы есть гораздо более простое решение: сокращение количества показателей и их вывод на более высокий уровень. Например, вместо того, чтобы оценивать разработку программного обеспечения по затратам, графику и прибавочной стоимости, попробуйте создать для проекта систему оценки счета прибылей и убытков, или прибыли на инвестированный капитал, и помогите группе использовать эти инструменты при принятии решений.

Рекомендация № 5: Найдите стимулы лучшие, чем деньги

Денежные премии могут быть мощным стимулом, однако обеспечиваемая ими мотивация не постоянна. Когда человек обладает нормальным доходом, мотивация приходит из других источников — достижения, рост, контроль, признание, прогресс и дружелюбное рабочее окружение. Какой бы хорошей ни была ваша система оценки и материального стимулирования, не рассчитывайте, что она обеспечит заоблачную эффективность в долгосрочной перспективе.

В своей книге «Скрытая ценность» Чарльз О'Рейли (Charles O'Reilly) и Джеффри Пфеффер (Jeffrey Pfeffer) представляют примеры компаний, добившихся превосходной эффективности от обычных людей. Эти компании выбрали ценности, ориентированные на человека, и совмещали их со своими действиями на всех уровнях. Они вкладывают деньги в людей, содействуют широкому обмену информацией,

опираются на групповую работу и делают упор на лидерстве, а не на руководстве. Наконец, они не используют деньги как главный стимул; эти компании выводят на первый план внутреннюю мотивацию: интересная работа, профессиональный рост, коллективизм, решение проблем и достижения.

К материальному стимулированию следует относиться как к взрывчатке: оно приводит к серьезным последствиям независимо от того, хотите вы этого или нет. Используйте деньги осторожно и аккуратно. Они могут создавать проблемы скорее, чем решать их. Стоит вам встать на путь материального стимулирования, и вы уже не сможете вернуться, даже когда этот путь станет неэффективным (а это со временем неизбежно). Убедитесь в том, что люди получают справедливую и адекватную компенсацию, а затем переходите к более эффективным способам повышения производительности.

Через полгода

У группы Сью снова праздник. Участники были удивлены, когда полгода тому назад вице-президент посетил их группу. Но они быстро справились с удивлением и сказали, что каждый хочет быть лучшим, что они хотят работать с лучшими, и им не нравится идея, что кто-то из них лучше других. Когда вице-президент ушел, команда поблагодарила Сью за поддержку и взялась за работу с новым энтузиазмом. И вот теперь, две версии спустя, клиенты демонстрируют признание их заслуг своими чековыми книжками.

В группе не было радикальных повышений зарплаты — только отдельные, заслуженные повышения в должности; тем не менее, компания расширила свою смету на обучение, а участники группы занялись обучением в других группах. Сью гордится ими, заполняя новые пересмотренные формы оценки, критерии которых лучше подходят для групп. На этот раз Сью уверена в том, что ее мнение не будет поставлено под сомнение.

Рик Шаут

MAC WORD 6.0

Появление Word 6.0 для Macintosh было встречено проклятиями. Обзоры были просто жуткими.

Как мне кажется, у Mac Word 6.0 были два недостатка: один настоящий, а другой политический.

Настоящая причина была в том, что программа медленно работала, содержала массу ошибок и требовала дополнительного обучения из-за отличий интерфейса от Mac Word 5.0. Но была и другая, политическая причина. Чтобы понять ее, необходимо вспомнить, что при первом появлении Macintosh был абсолютно революционным явлением. Графический интерфейс, шрифты, растровое представление экрана и умение издавать звук «бип!» так, как этого еще не делал ни один компьютер.

Многие издатели программного обеспечения не осознали новизну и превосходство Mac. Они решили, что это просто еще один компьютер. Поэтому они взяли свои старые DOS-программы и на скорую руку переделали их для Mac, сохранив смехотворный интерфейс текстового режима DOS, не обращая внимания на все значки, меню и все прочее. Портированные DOS-версии были ужасны, и компания испугалась, что при продолжении этой тенденции Mac лишится явных преимуществ своего графического интерфейса.

Поэтому евангелисты Apple вроде Гая Кавасаки (Guy Kawasaki) распространили среди правоверных последователей Mac мантру: Портирование — Это Плохо. В религии Mac существовал единственный истинный путь: написание «с нуля» абсолютно новой программы, использующей графический интерфейс. Компания Apple успешно приучала пользователей Mac отвергать любые приложения, полученные портированием DOS-версий текстового режима. К сожалению, обучение оказалось слишком успешным. Когда в результате портирования Word for Windows для Mac появился Word 6.0, в нем были меню, значки и прочие атрибуты продуктов Mac. Тем не менее, правоверные последователи Mac уловили, что происходит, и принялись петь свою мантру: «Портирование — Это Плохо», хотя портировалась программа для Windows, а не для DOS, а система Windows вообще представляла собой клон Mac.

Такова была политическая проблема. И хотя сейчас я подшучиваю над ней, это была вполне реальная проблема. Невозможно создать успешный продукт без

учета реальных и мнимых предубеждений его аудитории. Вот почему в наши дни компания Microsoft открыла в Кремниевой Долине новое подразделение, которое выпускает программы для Macintosh и всегда стремится включить в Mac-версию хотя бы одну возможность, отсутствующую в Windows-версии — это помогает привлечь на свою сторону поклонников Macintosh, невзирая на некоторую иррациональность. Крикливое сообщество Mac продолжает регулярно заявлять, что ему нужны «родные», написанные «с нуля» приложения для Mac, а не обычные портированные версии. Кстати говоря, эти разговоры приносят гораздо больше вреда, чем пользы — они отпугивают издателей, которые могли бы себе позволить портирование, но не располагают ресурсами для полноценного нового продукта... но это другая история.

Среди прочего, отчет Рика о Mac Word 6.0 напомнил мне о моей работе в Microsoft в начале 1990-х годов. Каждое решение принималось после долгих споров и размышлений очень, очень умными людьми, но, выражаясь метафорически, для галерки, швыряющейся огрызками, эти решения казались глупыми и некомпетентными. Так всегда бывает. Решения в области архитектуры сопряжены с чрезвычайно трудными компромиссами, особенно при ограниченной памяти и мощности процессора, поэтому так легко критиковать результат без понимания всех факторов, которые пришлось учитывать группе программистов. Поэтому журналисты, пишущие обзоры о наших продуктах, могут вполне законно жаловаться на поведение продукта, но любые попытки объяснить это поведение будут понятны только другому программисту. К счастью, некоторые участники этого процесса имеют льготные акции, и им принадлежит очень хороший дом на острове Мерсер, верхний этаж в деловой части Сиэтла, «Бентли» с откидным верхом и большая яхта. Так что не нужно их слишком жалеть. — *Ред.*

Мама часто говорила: «Когда бьешься головой о стену, в этом есть только один плюс: хорошо себя чувствуешь, когда прекращаешь».

Выпуск паршивого продукта чем-то сродни битью головой о стену — очень хорошо себя чувствуешь, когда потом в продолжение выпускаешь отличный продукт. С другой стороны, это заставляет тебя потратить немного времени на размышления о том, как не выпустить паршивый продукт снова.

Mac Word 6.0 был паршивым продуктом. И мы потратили время, разбираясь, как не повторить своей ошибки. За это время мы узнали кое-что новое, причем далеко не последним открытием было значение термина «в стиле Mac». Чтобы понять, почему Mac Word 6.0 был паршивым продуктом, необходимо понимать как историческую ситуацию, которая обусловила некоторые ключевые решения, так и технические проблемы, которые из этих решений следовали.

Mac Word 5 and Pyramid

В октябре 1991 мы выпустили Mac Word 5.0. Отзывы были просто блестящими. За свою работу мы получили аналог премии Тони[1] в области программирования для Mac: Mac World Eddy. Даже сегодня некоторые люди говорят, что Mac Word 5.0/5.1 был лучшей из когда-либо выпущенных версий Mac Word.

[1] См. http://en.wikipedia.org/wiki/Tony_Award — *Примеч. перев.*

Хотя Mac Word 5 был замечательным продуктом, у него была одна проблема: Win Word 2. Оба вышли примерно одновременно, но Win Word 2 обладал более широкими возможностями (прежде всего макроязыком[1], но были и другие). Это обстоятельство больше всего раздражало пользователей Mac Word. Они хотели равенства возможностей, и прямо сейчас! Чем дольше им приходилось ждать, пока Win Word сравняется с Mac Word, тем чаще нас поминали недобрыми словами.

Но у нас возникла проблема. Вернее, сразу две проблемы: первая — то, что Win Word и Mac Word строились по разным кодовым базам. Вторая — WordPerfect. На тот момент он все еще считался одним из основных конкурентов, и нам приходилось прикладывать определенные усилия, чтобы перегнать WordPerfect. Если бы мы продолжали разрабатывать Mac Word и Win Word на разных кодовых базах, Mac Word никогда бы не догнал Win Word по равенству возможностей.

В октябре 1991 у нас уже был разработан план решения первой проблемы: проект Pyramid. Мы решили полностью переписать Word — как для решения некоторых хронических проблем с несколько устаревшей к тому времени кодовой базой, так и для решения проблем с разделением кодовых баз. Как Win Word, так и Mac Word должны были строиться по одной кодовой базе.

Выходит Джефф Райкс, приходит Крис Питерс

Проблема равенства возможностей была решена. Впрочем, не совсем. В то же время Джефф Райкс получил повышение с поста начальника подразделения Word на какой-то другой пост в Microsoft (не помню точно, какой именно), а Крис Питерс получил повышение на его место. В те дни имя Джеффа Райкса было известно почти каждому. С другой стороны, Крис Питерс до перехода на Word работал руководителем по разработке Excel. Его любимым времяпрепровождением был боулинг, и он был известен тем, что весь его офис был завален пустыми банками из-под кока-колы.

Хотя Джефф Райкс считал проект Pyramid хорошей идеей, Крис Питерс взглянул на проблему WordPerfect и решил, что Pyramid не является хорошим способом решения проблемы равенства возможностей. Полное переписывание кода — дело рискованное. Его суть заключается в том, чтобы отступить на несколько шагов назад в краткосрочной перспективе, чтобы потом сделать более крупные шаги вперед в долгосрочной перспективе. Крис Питерс решил, что мы не можем себе позволить отступление, необходимое для Pyramid.

Итак, Крис Питерс угробил Pyramid. На этой стадии решить проблему равенства возможностей можно было только одним способом — начать Mac Word и Win Word с кодовой базы Win Word, что мы и сделали. Но это было не единственным последствием решения Криса Питерса. В то время руководителем по разработке Word был Крис Мейсон, который не согласился с решением Криса Питерса.

---

[1] WordBasic был на удивление полноценной средой программирования, позволявшей строить невероятно мощные приложения на базе полнофункционального редактора. — *Примеч. ред.*

В результате Крис Мейсон покинул группу Word и ушел на другую должность в Microsoft. Он понимал Mac и работал над разработкой Word еще со времен Mac Word 3.0. Пришедший на его место Эд Фрайс был «человеком Mac» в гораздо меньшей степени, чем Крис Мейсон, поэтому в группе руководителей Word высокого уровня была утрачена значительная доля понимания специфики Mac[1]. Невозможно точно сказать, к каким последствиям это привело, но с достаточно высокой вероятностью некоторые решения, принятые при работе над Mac Word 6.0, пошли бы в ином направлении.

Технические препятствия

Начиная с Win Word 2.0, кодовая база создавала пару технических проблем для тех из нас, кто работал на стороне Mac. Прежде всего, она была написана для Windows API. Решение проблемы не сводилось к простому написанию прослойки, эмулировавшей Windows API на Mac. В двух системах использовались принципиально разные механизмы работы с окнами, хотя интересно заметить, что новый Carbon API стал гораздо ближе к механизму Windows. Самая большая проблема заключалась в том, что в Windows существует концепция дочерних окон[2], а в Mac ее нет. Кроме того, в Windows все объекты являются субклассами объекта окна. Даже элементы управления представляют собой окна[3].

Еще одна проблема была обусловлена ограничениями Mac OS. Хотя 68K Classic Mac OS была хорошей операционной системой, у нее был очень серьезный недостаток: она плохо справлялась с управлением памятью. В сущности, она вообще почти ничего не делала для управления памятью. Пользователь должен был сообщить ОС, сколько памяти необходимо программе для работы, и именно столько памяти выделялось программе независимо от того, какой объем мог потребоваться в любой момент во время выполнения.

Проблема памяти усугублялась на компьютерах 68K, потому что выделенная программе память (независимо от настроек виртуальной памяти) должна была использоваться для хранения как кода, так и данных. В 68K код содержался в области, называвшейся «кодовым ресурсом». Кодовые ресурсы могли загружаться и выгружаться из памяти по мере надобности, а это означало, что фактические потребности программы в памяти радикально изменялись в зависимости от того, что хотел сделать пользователь.

Для примера возьмем подсистему проверки орфографии. Пользователь не будет проверять правописание постоянно, поэтому модуль проверки не обязан

---

[1] Эд Фрайс (Ed Fries) — выдающийся программист. Вероятно, его будут помнить прежде всего за изобретение концепции экранной заставки с рыбками, плавающими по экрану, хотя он проделал замечательную работу для Excel, Word и Xbox. — *Примеч. ред.*

[2] Дочерним называется окно, находящееся внутри другого окна. Например, в диалоговом окне с кнопками Yes/No обе кнопки являются дочерними окнами по отношению к диалоговому окну. — *Примеч. ред.*

[3] Элементами управления (controls) называются кнопки, текстовые поля, полосы прокрутки и т. д. — *Примеч. ред.*

постоянно находиться в памяти. Но модуль проверки орфографии — не простой фрагмент кода, а настоящий пожиратель памяти. Механизм работы с памятью в 68K Classic Mac OS означал, что для загрузки огромного модуля проверки правописания приложению необходимо было выделить минимальный объем памяти.

Я провожу различия между 68K Classic Mac OS и PowerPC Classic Mac OS, потому что на PowerPC компания Apple изменила механизм хранения, загрузки и исполнения кода. Для тех, кто еще помнит, при вызове Get Info для приложения показывались две разные величины требований к памяти: с включенной виртуальной памятью и без нее. При включенной виртуальной памяти работа с кодом приложения могла осуществляться через механизм виртуальной памяти «с подгрузкой по требованию», поэтому код не обязан был помещаться в раздел памяти приложения. Пресловутый модуль проверки орфографии можно было не учитывать при вычислении минимальных требований приложения к памяти.

Впрочем, я не стану перекладывать всю вину на Apple. Полноценная реализация виртуальной памяти требует аппаратной поддержки. В 1984 году микропроцессоры не обладали полной функциональностью, необходимой для полной реализации виртуальной памяти с подгрузкой по требованию, так что ее поддержка в исходной версии Mac OS была бы напрасной тратой времени. Мы часто принимаем архитектурные решения, абсолютно осмысленные в свете текущих системных ограничений — только для того, чтобы они начали преследовать нас, когда производительность систем по закону Мура возрастает на порядок. Отказ Apple от линейки процессоров Motorola 68K в пользу PowerPC объяснялся несколькими причинами, не последней из которых была возможность пересмотреть некоторые из принятых ранее решений.

Технические достижения

Благотворное воздействие закона Мура позволяет нам запросто устанавливать 128 Мбайт или даже полгигабайта памяти в портативный компьютер, поэтому многим будет нелегко понять, насколько серьезные проблемы для Mac Word 6.0 создавала работа с памятью в 68K. Но мы все же попытались заставить всю программу работать в 4 Мбайт памяти — причем речь идет обо всей системной памяти, не только о разделе приложения.

Проблема была довольно серьезной. Word 6 по новым возможностям значительно превосходил Win Word 2.0. По сравнению с Mac Word 5.0 изменений хватило бы на два выпуска. OLE[1]; встроенный лексический анализатор и механизм принятия решений на базе правил, необходимый для работы автозамены/автоформата; модуль проверки грамматики, включавший уникальную технологию обработки естественного языка (из-за которой подсистема проверки грамматики занимала еще больше памяти), в сочетании с такими вещами, как полноценный

[1] OLE (Object Linking and Embedding) — механизм, позволяющий внедрять документы внутрь других документов (скажем, графику или фрагмент электронной таблицы в документ текстового редактора). — *Примеч. ред.*

макроязык (WordBasic) — все эти факторы делали Mac Word 6.0 слишком большим относительно возможностей стандартных систем Mac того времени.

Обратите внимание на уточнение «относительно». Чтобы представить картину в целом, запустите BBEdit на компьютере с Mac OS X, откройте окно терминала и введите «top» в командной строке. Прочитайте значения в столбцах RSIZE и VSIZE. Когда я открываю файл .tcshrc в BBEdit, эти значения равны 12.1 и 164 Мбайт соответственно. А когда я набираю этот документ в новейшей версии Word 2004, эти значения равны 36,6 Мбайт и 222 Мбайт соответственно.

Самое невероятное, что нам все же удалось заставить Word 6.0 работать в системе, имевшей всего 4 Мбайт памяти (пусть неторопливо, но все же работать). Чтобы в полной мере осознать масштаб наших достижений, необходимо кое-что знать о том, как пишутся и выполняются программы. Далее я попытаюсь популярным языком разъяснить довольно сложную техническую проблему. Если у вас голова пойдет кругом, переходите к следующему разделу.

Программы пишутся в виде относительно небольших фрагментов кода, называемых «функциями». Каждая функция представляет отдельный функциональный аспект программы. Функции могут представлять высокоуровневые концепции (например, форматирование целой страницы текста) или низкоуровневые концепции (форматирование отдельной строки текста на странице). Для выполнения своей работы высокоуровневые функции вызывают низкоуровневые функции; существует специальный протокол, при помощи которого компьютер узнает, как вернуть управление из низкоуровневой функции в высокоуровневую функцию, из которой он был вызван. В этом протоколе, называемом «процедурными прологом и эпилогом», задействован так называемый «стек вызовов». При работе низкоуровневого кода высокоуровневый код помещается в стек вызова.

Попытка выполнения фрагмента кода в области памяти, размер которой меньше размера кода, приводит к *выгрузке кода* из памяти. Обычно выгрузка реализуется очень просто, если выгружаемый код не переходит с высокого уровня на низкий. Хорошим примером служит модуль проверки орфографии. Он представляет обособленную функциональную единицу, и если он не понадобится в ближайшее время, его можно выгрузить из памяти, не беспокоясь о его повторной подгрузке при завершении выполнения текущего фрагмента кода.

Однако группировка кода также может осуществляться на уровне, пересекающем границы высокого/низкого уровня. Например, код форматирования страницы текста может принадлежать одному модулю (сегменту кода), тогда как код форматирования одной строки находится в другом модуле. При форматировании одной строки текста вам не нужно, чтобы код форматирования всей страницы постоянно находился в памяти. Во время работы кода форматирования строки код форматирования страницы может быть выгружен из памяти (по крайней мере, на концептуальном уровне).

И здесь возникает проблема: код форматирования страницы вызывает код форматирования строки; это означает, что код форматирования страницы остается в стеке вызовов. При завершении кода форматирования строки протокол, используемый компьютером для возврата управления коду форматирования страницы, должен знать, что код форматирования страницы был выгружен из памяти. Проблема была настолько сложной, что в документации Apple утверждалось, что

выгрузка кода, находящегося в стеке вызовов, в принципе невозможна. И все же именно это нам удалось сделать в Word 6.0.

У выгрузки кода, находящегося в стеке вызовов, имеется одна неприятная оборотная сторона — иногда она приводит к явлению, называемому пробуксовкой (thrashing). Возьмем наш пример с форматированием страниц/строк. Форматирование страницы основано на вызове функции форматирования строки для каждой строки текста на странице. При каждом пересечении границы между кодом форматирования страницы и форматирования строки необходимо остановиться и загрузить в память фрагмент кода. А это, в свою очередь, приводит к необходимости выгрузить из памяти другой фрагмент кода.

Чтобы объяснить суть происходящего, я очень сильно упростил весь процесс. При выборе выгружаемых частей программы для освобождения памяти, необходимой для подгрузки непосредственно используемого кода, алгоритм выгрузки действует более умно. Таким образом, на практике пробуксовка между кодом форматирования страницы и строки крайне маловероятна. Тем не менее, при достаточно малом объеме доступной памяти пробуксовка все же случается. В таких ситуациях быстродействие системы падает ниже плинтуса.

Смысл термина «в стиле Mac»

Итак, Mac Word 6.0 был слишком большим и медленным для памяти, которой оснащалось большинство компьютеров того времени. Но не по этой причине Mac Word был таким паршивым продуктом... по крайней мере, не напрямую. Вскоре после выхода Word 6.0 многие пользователи смогли себе позволить установку дополнительной памяти для дополнительных возможностей Mac Word 6.0. По их оценке, новые возможности были очень полезными, а расходы на расширение памяти окупались повышением производительности труда. Закон Мура и PowerPC решили бы эту проблему в надлежащее время. Более того, хотя многие жаловались на быстродействие, чаще всего нам приходилось слышать жалобы на то, что Mac Word 6.0 не был сделан «в стиле Mac». Из-за этого мы потратили много времени, выясняя, что же люди имели в виду под «стилем Mac». Одни проводили исследования в целевых группах, другие вступали в группы Usenet. Мы беседовали с авторами обзоров и друзьями, которые пользовались продуктом. Оказалось, что «в стиле Mac» значит «похожий на Mac Word 5.0».

Мы потратили столько времени и усилий на решение технических проблем Mac Word 6.0, что не успели согласовать интерфейс Mac Word 6.0 с интерфейсом Mac Word 5.0. В результате между способами выполнения одних и тех же операций в Mac Word 5.0 и Mac Word 6.0 существовало множество различий — мелких, серьезных, а порой и просто неоправданных. Появившийся в конечном итоге интерфейс был весьма неуклюжим по сравнению с элегантностью Mac Word 5.0. Что еще важнее, в некоторых областях пользователям Mac Word пришлось забыть обо всем, что они знали, и изучать заново, как те же операции выполняются в Word 6.0. Мой любимый пример — определение стилей. В Mac Word 5.0 определение стиля было полумодальной операцией; стили определялись и модифицировались по аналогии с изменением шрифтов и свойств абзацев в самом документе.

В Mac Word 6.0 операция стала полностью модальной. Многочисленные меню и кнопки панелей инструментов, использовавшиеся в Mac Word 5.0 (хорошо знакомые пользователям), были заменены одним раскрывающимся меню в диалоговом окне New/Modify Style. Даже сегодня для смены шрифта или информации абзаца в стилях Word 2001 и Word X невозможно использовать палитру форматирования; это одна из тех вещей, которые мне хотелось бы исправить в Word перед уходом из подразделения Mac в Microsoft.

Выясняя смысл выражения «в стиле Mac», мы также поняли другое — мы не сможем разрабатывать продукты «в стиле Mac», если Office останется единым продуктом, на базе которого будут строиться версии для Win и Mac. Сам факт возникновения проблемы «стиля Mac» означал, что между рынками Win Word и Mac Word существуют фундаментальные различия. Если мы собирались понять оба рынка, нашим продуктам для Mac и Win потребуются раздельные маркетинговые отделы и группы руководства. И именно благодаря некоторым урокам, усвоенным из Mac Word 6.0, подразделение Mac все еще продолжает существовать.

Время от времени нам все еще приходится биться головой о стену — ведь понимание потребностей пользователей нельзя отнести к точным наукам. Но зато это происходит гораздо реже, чем раньше. И мы действительно чувствуем себя гораздо лучше.

Клей Ширки

Группа как собственный худший враг

В конце 1980-х годов в разработке программного обеспечения произошел серьезный перелом. Примерно до 1985-го года главной целью программы считалось решение поставленной задачи любыми необходимыми средствами. Нужно набивать входные данные на перфокартах? Ничего страшного. Из-за ошибки в одной карте приходится выбрасывать всю проделанную работу и начинать заново? Нет проблем. Человек склонялся перед машиной, как Чарли Чаплин в «Новых временах».

Но с появлением персональных компьютеров планка неожиданно поднялась. Одного решения проблемы стало недостаточно: проблема должна была решаться *легко*, с учетом типичных человеческих слабостей. Например, клавиша Backspace компенсировала нашу невнимательность, не говоря уже о всевозможных окнах, значках, меню и многоуровневой отмене. Люди назвали это «практичностью», и было это хорошо.

Но когда отрасль разработки программного обеспечения попыталась нанимать экспертов, выяснилось, что это совершенно новая область, и ей никто не занимается. В психологии существовала своя ниша, называемая эргономикой, но она в основном была сосредоточена на объектах материального мира — скажем, на определении оптимальной высоты офисного кресла.

В конечном счете, практичность выделилась в самостоятельную область исследований с последователями-самоучками и университетскими курсами, и ни один программный проект не мог считаться полным без хотя бы беглого взгляда в сторону практичности.

Сейчас нас ждет аналогичный перелом.

Сразу же после пришествия Интернета программы перестали ассоциироваться исключительно с взаимодействиями «человек-компьютер» и начали постепенно смещаться в область взаимодействий «человек-человек». Появились новые приложения: Web, электронная почта, системы диалогового обмена (Интернет-пейджеры) и форумы — все они были направлены на общение людей друг с другом при содействии программ.

И вдруг выяснилось, что при создании программы недостаточно думать только о физической возможности коммуникаций; нужно было также позаботиться о том, чтобы коммуникации были социально успешными. В эпоху практичности проектировщику приходилось принимать технические решения, облегчающие

использование программы массовой аудиторией; в эпоху социального программного обеспечения ему приходилось принимать решения, направленные на выживание и процветание социальных групп, и достижение их целей даже в том случае, если эти цели противоречили целям отдельных индивидов. Дискуссионная группа, разработанная экспертом в области практичности, может оптимизироваться для максимального упрощения отправки сообщений — например, спама о виагре. Но при проектировании социального программного обеспечения становится ясно, что некоторые вещи должны затрудняться, а не упрощаться, и если вам удастся сделать публикацию спама практически невозможной — значит, вы успешно справились со своей задачей. Функциональность проектируется в расчете на успешную деятельность группы, а не ее отдельных членов.

В наши дни вряд ли кто-нибудь серьезно занимается проектированием программного обеспечения, рассчитанного на взаимодействия между людьми. Область проектирования социального программного обеспечения еще находится на начальной стадии. Более того, мы еще даже не достигли той точки, в которой разработчики осознают необходимость учета социологии и антропологии тех групп, которые будут пользоваться этими программами, объединят свои усилия и будут удивляться социальным взаимодействиям, развивающимся на базе их программ.

Клей Ширки стал одним из первопроходцев в этой области, а его доклад «Группа как собственный худший враг» запомнится как переломный момент в повсеместном осознании того факта, что в новую эпоху социология и антропология занимают в проектировании программного обеспечения такие же ключевые позиции, как и практичность в прошлом. — *Ред.*

Всем доброе утро. Сегодня я хочу поговорить о социальном программном обеспечении и о некоторых закономерностях, которые я неоднократно наблюдал в социальных программах для поддержки больших долгосрочных групп. В частности, речь пойдет о том, что мне сейчас представляется одной из главных проблем при проектировании крупномасштабного социального программного обеспечения — о закономерности, вынесенной в название доклада: «Худшим врагом группы является сама группа».

Позвольте мне привести определение социального программного обеспечения, потому что этот термин все еще остается достаточно туманным. Мое определение предельно просто: это программа, обеспечивающая взаимодействие в группах. Мне также хотелось бы подчеркнуть радикальность концепции социального программного обеспечения, несмотря на простоту определения. В Интернете встречается множество коммуникационных схем, главным образом «точка-точка», двусторонние исходящие «один ко многим» и двусторонние «многие ко многим».

До появления Интернета также существовали различные схемы поддержки двусторонних взаимодействий «точка-точка». У нас был телефон и телеграф. Мы были знакомы с технологическими аспектами такого общения.

До появления Интернета существовало множество схем поддержки односторонней широковещательной рассылки информации. Я мог разместить свою информацию на телевидении или на радио, опубликовать ее в газете. В нашем распоряжении был печатный станок. Следовательно, хотя Интернет вносит свой положительный вклад в коммуникационные схемы этого вида, технологическая поддержка взаимодействий «точка-точка» и широковещательных схем существовала и до Интернета.

С программным обеспечением для групп дело обстоит иначе. До появления Интернета последним технологическим объектом, который оказывал сколько-нибудь заметное влияние на совместные посиделки и беседы людей, был стол. Самым близким аналогом можно назвать конференцсвязь, которая все равно никогда не работала нормально: «Алло? Нужно нажать вот эту кнопку? Вот черт, почему-то связь разорвалась». Устроить конференцию по телефону трудно, зато разослать по электронной почте пять сообщений с вопросом «Ну что, идем есть пиццу?» проще простого. До смешного простое формирование группы стало новой закономерностью, тогда как раньше простых технологий для решения этой задачи никогда не было.

Социальное программное обеспечение существует не более 40 лет, начиная с Plato BBS, а широкое распространение оно получило только лет 10 назад или около того, так что мы еще только ищем возможные пути. Мы еще учимся использовать такие программы.

Однако определение с «обеспечением взаимодействия в группах» во многих отношениях принципиально неудовлетворительно, потому что оно не ссылается на конкретный класс технологий. Для примера возьмем электронную почту: она очевидным образом поддерживает социальные взаимодействия, но при этом также может поддерживать широковещательную схему. Скажем, спамер может отправить свое сообщение миллиону людей, однако эти люди не собираются общаться друг с другом, а спамер не собирается общаться с ними — спам является электронной почтой, но он не социален. Если я отправляю сообщение вам, а вы отвечаете мне, у нас идет общение по схеме «точка-точка», но не такое, которое способствует созданию групповой динамики.

Таким образом, электронная почта иногда поддерживает социальные схемы, а иногда — нет. То же относится и к блогам. Скажем, если я — Глен Рейнольдс с Instapundit.com[1], я публикую свои материалы для миллиона пользователей в месяц и отключаю комментарии в своем блоге, то результат представляет собой широковещательную рассылку — пользователи Глена не могут ему ответить и не общаются друг с другом. Конечно, очень интересно, что Глен обращается к такой массе как отдельная личность, и все же такая схема ближе к MSNBC, нежели к общению в традиционном понимании. С другой стороны, если речь идет о скоплении из десятка пользователей LiveJournal[2], беседующих друг с другом «о жизни» — это уже социальное общение. Таким образом, блоги не обязательно социальны, хотя они могут поддерживать социальные схемы.

Хотя это определение («программы, обеспечивающее взаимодействие в группах») пересекает границы существующих категорий, я все же считаю его правильным, потому что оно учитывает изначально социальную природу проблемы. Группы возникают «на стадии выполнения». Нельзя предсказать заранее, как будет действовать любая конкретная группа, поэтому невозможно предусмотреть в программе все возможные сценарии. Написано немало литературы, в которой говорится: «Мы создали эту программу, пришла группа и начала ее использовать.

---

[1] Популярный политический блог. — *Примеч. ред.*

[2] Сетевая служба личных журналов — фактически те же блоги, но с большим уклоном в сторону формирования сообщества. — *Примеч. ред.*

При этом проявились некоторые аспекты поведения, безмерно нас удивившие, поэтому мы решили их документировать». Этот шаблон встречается снова и снова. (Я слышу смех Стюарта[1]. WELL был как раз одним из мест.)

Разобравшись с основами, я разделю остаток доклада на три части. Лучшее объяснение явлений, возникающих при взаимодействии групп людей, встретилось мне в психологическом исследовании, написанном еще до появления Интернета. Итак, первая часть будет посвящена исследованиям У. Р. Биона — которые, как мне кажется, помогают объяснить, почему группа является собственным худшим врагом.

Часть вторая: почему именно теперь? Что сейчас происходит такого, из-за чего нам нужно думать на эту тему? Мне кажется, что в настоящее время мы наблюдаем революцию в области социального программного обеспечения, и это действительно интересное явление.

В третьей части я хочу выделить некоторые свойства (числом около полудюжины), которые, на мой взгляд, являются основополагающими для любых программ, на основе которых формируются большие, долгосрочные группы.

Часть 1: Почему группа является собственным худшим врагом?

Итак, часть первая. Лучшее объяснение проявлений этого феномена — «группы как собственного худшего врага» — встретилось мне в книге У. Р. Биона «Психология групп», написанной в середине прошлого века.

Бион был психологом и проводил групповую терапию в группах невротиков (проведение параллелей с Интернетом остается читателю для самостоятельной работы). Пытаясь вылечить своих пациентов, Бион понял, что в совокупности они представляли собой группу, нацеленную на борьбу с его терапией. Между ними не существовало явного общения или координации. Но каждый раз, когда он пытался сделать нечто, способное возыметь эффект, группа каким-то образом гасила его начинание. А он тем временем сходил с ума (в житейском смысле слова), пытаясь определить, нужно или нет рассматривать ситуацию под следующим углом: «Действуют ли эти индивиды самостоятельно или представляют собой скоординированную группу?»

Найти ответ Биону так и не удалось, поэтому он решил, что сама неразрешимость проблемы является ответом. На вопрос «Какое поведение характерно для групп людей: поведение совокупности личностей или поведение слитной группы?» Бион отвечает, что группы людей «безнадежно привержены обоим». Другими словами, они безнадежно привержены к проявлениям, характерным как для отдельных личностей, так и для участников групп.

Он говорит, что люди в своей основе индивидуальны, и также изначально социальны. У каждого из нас имеется своего рода рациональный механизм принятия решений, который позволяет нам оценивать происходящее, принимать решения

Стюарт Брэнд (Stewart Brand) из WELL, одного из первых сетевых сообществ, предшественников Интернета; теперь часть Salon.com. — *Примеч. ред.*

и действовать на их основе. И мы также способны устанавливать внутренние эмоциональные связи с другими группами людей, превосходящими интеллектуальные аспекты индивида.

Бион был настолько уверен в своей правоте, что поместил на обложку книги изображение «куба Неккера» — одну из тех картинок, на которой можно рассмотреть два разных изображения, но в любой момент видно только одно. Таким образом, группа может анализироваться и как совокупность индивидов, и как носитель групповых эмоциональных проявлений.

В группах с формальными признаками принадлежности — то есть в группах с четкими обозначениями вида «Я вхожу в такую-то гильдию в такой-то многопользовательской ролевой игре» — формирование групповых связей выглядит абсолютно естественно. Но тезис Биона состоит в том, что этот эффект возникает гораздо, гораздо глубже, и начинает действовать гораздо, гораздо быстрее, чем предполагает большинство из нас. Поэтому я хочу пояснить сказанное примером, а чтобы пояснение получилось более наглядным — воспользуюсь историей из вашей жизни. И хотя я вас совершенно не знаю, но могу совершенно точно утверждать, что с вами случалось нечто подобное.

Вы находитесь на вечеринке, и вам скучно. Вы говорите: «Как же мне все надоело. Я бы предпочел находиться в другом месте. Например, просто поспать дома. Здесь даже поговорить не с кем». Невесть по какой причине вечеринка не вызывает у вас интереса. И здесь происходит нечто весьма интересное: вы не уходите. Вы принимаете решение «Мне здесь не нравится». В книжном магазине вы бы сказали: «С меня хватит» и ушли. В кофейне вы бы сказали: «Мне скучно» и ушли. Однако вы сидите на вечеринке, решаете: «Мне здесь не нравится, я не хочу здесь находиться»... и не уходите. Так вот, Бион говорит именно о таких социальных привязанностях.

А затем происходит еще одно интересное явление. Через 20 минут один человек встает и одевается, и что происходит? Внезапно все начинают одеваться одновременно. А это значит, что все решили, что вечеринка не задалась, но никто ничего не сделал, пока некоторое инициирующее событие не «выпустило воздух» из вечеринки, и все вдруг почувствовали, что им можно расходиться. Этот эффект настолько распространен, что его иногда называют «парадоксом групп». Совершенно очевидно, что групп без участников не бывает. Менее очевидно другое: не бывает участников без групп — иначе в чем вы участвуете?

Таким образом, формирование группы представляет собой очень сложный момент: достаточное число индивидов по какой-то причине соглашаются, что происходит нечто стоящее, и принимают решение: «Это хорошо, и это нужно сохранить». И в этот момент, пусть на подсознательном уровне, начинают проявляться групповые эффекты. Именно эти эффекты мы наблюдаем снова и снова в сетевых сообществах.

Далее Бион решил, что эффект, наблюдаемый с невротиками, представлял собой защиту группы от его попыток заставить группу делать то, что, по его мнению, ей следовало делать. Группа пациентов проходила терапию для улучшения своего состояния, однако в процессе терапии они противодействовали тем самым мерам, которые должны были им помочь. После многолетних наблюдений Бион выделяет несколько конкретных шаблонов поведения, проявляемых группой для

противодействия основной цели ее формирования, то есть терапии. Он описывает три таких шаблона.

Первый шаблон — это разговоры о сексе. Группа воспринимает свое предназначение как ведение флирта или фривольных разговоров, или обмена эмоциями между парами участников.

Представьте, что вы заходите на IRC (Internet Relay Chat) и просматриваете список имен каналов. При виде канала #hamradio channel напрашивается мысль: «Понятно, здесь говорят о любительском радио». Но присоединившись к каналу, вы почти неизменно обнаруживаете, что речь идет о сексе, обычно на уровне пикантных двусмысленностей. Согласно Биону, тема секса всегда присутствует в человеческих разговорах; интересно заметить, что в асинхронных коммуникациях вроде списков рассылки этот шаблон проявляется гораздо реже, чем в синхронных средах вроде IRC. Это один из базовых шаблонов, на которые всегда может переключиться группа, чтобы подменить сложную исходную цель более простой целью.

Второй базовый шаблон, описанный Бионом, — выявление и поношение внешних врагов. Этот шаблон также получил очень широкое распространение. Каждый, кто участвовал в движении Open Source в середине 1990-х, мог наблюдать его постоянно. Тех, кто действительно стремился к продвижению Linux на настольные системы, ждал большой список необходимых дел. Но вместо этого всегда можно поговорить о Microsoft и Билле Гейтсе. И люди так распалялись, что у них начинал бить из ноздрей пар.

Казалось, в те времена у движения Open Source практически не было врагов, что объяснялось его подходом к работе: если вы хотели что-то улучшить, вам предлагался список дел. Берите и улучшайте. Но вместо этого всегда можно было с пеной у рта поносить Microsoft и Билла Гейтса.

Ничто не возбуждает группу так, как внешний враг. Следовательно, даже если кто-то в действительности не является вашим врагом, указание на его враждебность создает приятное чувство сплоченности группы. И группы часто тяготеют к самым параноидальным участникам и делают их своими лидерами, потому что эти люди лучше всего справляются с поисками внешних врагов.

Третий шаблон, выделенный Бионом, — религиозное поклонение, то есть выявление и почитание религиозного символа («иконы») или набора религиозных догматов. В сущности, религиозный шаблон подразумевает создание чего-то, не подверженного критике. С его проявлениями в Интернете мы сталкиваемся сплошь и рядом. Зайдите на форум или в конференцию, посвященную Толкиену, и попробуйте сказать: «Знаете, а "Две башни" скучноваты. Такая длиииииинная книга. И все эти описания пересечения леса не так уж нужны, потому что это все равно один и тот же лес».

Попробуйте вступить в такую дискуссию. На «двери» группы говорится: «Для обсуждения работ Толкиена». Войдите и попробуйте обсудить.

Возможно, где-то скажут: «Да, но это было необходимо, чтобы передать чувство утомления героев», или что-нибудь в этом роде. Но в большинстве мест вас просто зафлеймят на все корки, потому что вы посягнули на религиозный текст. В группах часто имеется небольшой набор основных догматов, верований или интересов, стоящих вне критики, потому что они способствуют объединению

группы. Даже в группах, основанных для интеллектуального общения, при посягательстве на эти базовые верования на первый план выходят эмоции — потому что вы не просто предлагаете свое мнение, а угрожаете целостности группы.

Шаблоны Биона проявляются в Интернете не потому, что они заложены в программы, а потому, что Интернетом пользуются люди. Бион лишь выявил возможность группы подавлять свои сложные цели этими базовыми побуждениями. И в конечном счете он пришел к необходимости структуры в группах. Необходимы своды правил. Необходима регламентация. Нормы, ритуалы, законы — короче, любые способы, которые позволяют выделить из бесконечной Вселенной линий поведения относительно узкий круг приемлемых. Все это удерживает группы от перехода к шаблонам-заменителям и подмены ими действительного результата. Каждый, кто работал в конкурентных отраслях, знает, как легко угробить двухчасовое совещание, упомянув о замыслах конкурентов. В этот момент все перестают думать о том, что для достижения целей необходимо основательно потрудиться, и начинают попеременно поносить конкурентов и убеждать себя в отсутствии какой-либо угрозы.

Но самое важное, по словам Биона, заключается в том, что различные формы групповых структур, созданных за века, были необходимы для защиты группы от самой себя. Структура группы существует для того, чтобы группа оставалась сконцентрированной на исходной цели и не уклонялась от нее, скатываясь к базовым шаблонам. Структура группы защищает саму группу от действий ее членов.

Этот шаблон также постоянно встречается в сетевых сообществах. В 1970-х годах была открыта BBS Communitree — одна из самых первых BBS с модемным доступом. Она была открыта в те времена, когда у людей еще не было своих компьютеров — компьютеры были только в учреждениях.

Работа Communitree базировалась на принципах открытого доступа и свободного диалога (Communitree — разве это не звучит как «Калифорния 70-х»?). Идея состояла в том, чтобы отказаться от любых структур, и тогда сами собой возникнут новые прекрасные социальные схемы.

В самом деле, такое действительно случается — это известно каждому, кто внедряет социальное программное обеспечение в группах, ранее не подключенных к Сети. Происходят невероятные вещи. На ранней стадии существования Echo[1] или Usenet, в начале проекта Lucasfilm Habitat (одна из первых многопользовательских игр), снова и снова вы видите невероятный энтузиазм людей, которые вдруг получили возможность общаться между собой так, как это было невозможно раньше.

Но не все так безоблачно; постепенно возникают проблемы. В этом конкретном случае одна из проблем была обусловлена тем фактом, что одним из учреждений, подключенных к Communitree, была средняя школа. А кто в 1978 году ошивался в комнатах с компьютерами и модемами? Конечно, старшеклассники. Эти ребята не слишком интересовались умными взрослыми разговорами. Их больше интересовали неприличные шуточки и сальные разговоры. Им нравилось беситься, захламляя весь форум нецензурными словечками и бессмысленной болтовней.

---

[1] Маленькое, но престижное сетевое сообщество из Нью-Йорка, созданное Стейси Хорн из Нью-Йоркского университета; один из предшественников Интернета.

Взрослые, создавшие Communitree, были в ужасе из-за нашествия подростков. Место, основанное для открытого доступа, оказалось *слишком* открытым. Они не могли защититься от собственных пользователей. Место, предназначенное для свободы слова, оказалось слишком свободным. Нельзя было сказать: «Нет, это не та свобода слова, которую мы имели в виду».

Получалось, что для защиты от нашествия приходилось делать то, что противоречило всей идеологии. В конечном счете проект был просто закрыт.

Спрашивается, была ли неспособность основателей к самозащите технической или социальной проблемой? Разве программное обеспечение не позволяло решить проблему? Или дело было в социальной конфигурации группы, основавшей Communitree, которая просто не могла переварить идею введения цензуры для защиты системы? Впрочем, это не столь важно, потому что технические и социальные аспекты тесно переплетаются друг с другом, и полностью разделить их невозможно.

Печальный опыт Communitree учит нас, что по-настоящему опасно нападение изнутри. Работу Communitree парализовали не внешние злоумышленники, пытавшиеся устроить сбой на сервере. Сервер был закрыт из-за того, что люди входили на него и отправляли сообщения — то есть делали именно то, для чего предназначалась система. Технологические схемы нормального использования и атаки на машинном уровне настолько близки, что на технологическом уровне просто невозможно указать, что должно, а что не должно происходить в системе. Одни пользователи хотели, чтобы система продолжала существовать и предоставляла форум для ведения дискуссий. А другие пользователи, школьники-старшеклассники, либо оставались равнодушными, либо занимали активно враждебную позицию. И система не давала возможности первой группе защититься от второй.

Схема повторялась снова и снова. Кто-то строил систему, подразумевая некоторые стереотипы поведения пользователей. Пользователи приходили и демонстрировали другое поведение. Люди, управлявшие работой системы, к своему ужасу обнаруживали, что технологические и социальные аспекты фактически оказывались неразделимыми. История повторялась много раз. На самом деле наблюдать за ее повторением очень печально; хотя на эту тему написана масса документации, люди, начинающие подобные проекты, обычно не читают эти отчеты.

Все эти повторения можно было бы снисходительно охарактеризовать поговоркой «на ошибках учатся», но учиться на ошибках — худший способ что-либо узнать. Конечно, этот способ чуть лучше тупого запоминания, и все же он не идеален. Лучший способ узнать что-нибудь — это когда кто-то другой узнает об этом и говорит вам: «Не лезь в воду, здесь водятся аллигаторы».

Узнавать о присутствии аллигаторов на собственном опыте, мягко говоря, неприятно — гораздо хуже, чем прочитать об этом. К сожалению, в этой области чтение не принесло особой пользы. По этой причине очерк «Уроки Lucasfilm Habitat»[1] от 1990 года во многом похож на историю Communitree, написанную Роуз.

Существует замечательный документ, который называется «LambdaMOO идет в новом направлении». Он посвящен мастерам LambdaMOO, эксперименту Павла

[1] См. http://www.fudco.com/chip/lessons.html.

Кертиса (Pavel Curtus) из Xerox PARC по созданию MUD[1]. Однажды мастера LambdaMOO объявили: «Мы запустили эту систему, и теперь в ней начали происходить всевозможные интересные социальные эффекты. В дальнейшем мы, мастера, будем участвовать только в технологических аспектах. Дела социальные нас не касаются».

Позднее (кажется, через полтора года) мастера вернулись в крайне раздраженном состоянии. И они сказали: «Из вечного нытья пользователей мы поняли одно: мы не можем сделать то, что обещали. Мы не можем отделить технологические аспекты управления виртуальным миром от социальных. Поэтому мы возвращаемся, снова садимся за руль и будем управлять системой. Фактически мы назначаем себя правительством, потому что этому месту необходимо правительство. Без нас здесь все разваливается».

Люди, работающие над социальным программным обеспечением, по духу ближе к экономистам и политологам, нежели к разработчикам компиляторов. Вроде бы речь идет о том же программировании, но когда одним из факторов времени выполнения являются группы людей, ситуация в корне меняется. В области политики подобные кризисы называются конституционными кризисами. Они возникают, когда противоречия между индивидуумом и группой, а также правами и обязанностями индивидуумов и групп, становятся настолько серьезными, что с ними нужно что-то делать.

И самым серьезным кризисом оказывается самый первый, потому что одних правил недостаточно. Вам потребуются правила для определения этих правил. И мы снова встречаемся с проявлениями этого феномена в крупных социальных программных системах с длительным сроком жизни. Система правил является необходимым компонентом всех крупных долговечных гетерогенных групп.

Джефф Коэн (Geoff Cohen) прекрасно высказался на эту тему. Он сказал: «Вероятность того, что в любой немодерируемой группе рано или поздно возникнет флейм по поводу необходимости модератора, со временем приближается к единице». По мере того, как группа осознает свое существование в качестве таковой, она с очень высокой вероятностью начнет требовать введения дополнительной структуры для защиты себя от самой себя.

Часть 2: Почему сейчас?

Если то, о чем я говорю, так часто происходило в прошлом и продолжает происходить сейчас, если все это было документировано, а литература по психологии была написана еще до появления Интернета, то что же сейчас происходит такого, из-за чего эта тема стала такой важной? Я не могу сказать точно, что именно, но наблюдения показывают, что сейчас идет революция в области социального программного обеспечения. Количество людей, пишущих программы для обеспечения или усовершенствования взаимодействия в группах, просто потрясает. Вот уже лет шесть или восемь Web заставляет нас уделять первостепенное внимание количественным

[1] MultiUser Dungeon, текстовая сетевая приключенческая игра. — *Примеч. ред.*

характеристикам. Причин было много: и слабое связывание, отсутствие хранимого состояния, и безумное масштабирование, и всеобщее стремление разрастись до максимально возможных размеров. «Сколько пользователей у Yahoo? Сколько клиентов у Amazon? А сколько читателей у MSNBC»? И во всех случаях ответ один: «Много!» Но перед MSNBC стоит более простая задача: им не нужно было придумывать, как дать читателям возможность общаться друг с другом.

Недостаток увлечения размерами и масштабами заключается в том, что плотная схема с тесным переплетением, обеспечивающая общение и взаимодействие в группах, не может поддерживаться в сколько-нибудь крупном масштабе. В малых сообществах дело обстоит иначе — в малых группах возможны различные взаимодействия, недоступные для больших групп, и за годы существования Web мы просмотрели этот интересный уровень малых групп. В группах численностью более десятка, но менее нескольких сотен, возможны формы общения, которые не могут поддерживаться при общении с тысячами и миллионами пользователей в одной группе.

Долгое время у нас были списки рассылки и BBS. Недавно появились системы диалогового обмена (Интернет-пейджеры), и в течение некоторого времени этим дело и ограничивалось. Но сейчас внезапно начали появляться новые формы, включая блоги и Wiki, и, как мне кажется, еще важнее — стали появляться новые платформы. Мы получаем RSS, мы получаем общие объекты Flash. В нашем распоряжении оказываются средства быстрого конструирования на базе некоторых инфраструктур, доступность которых можно считать само собой разумеющейся, и это позволяет очень быстро опробовать новые возможности. Я говорил со Стюартом Баттерфилдом (Stewart Butterfield) о Flickr — новом приложении, разработкой которого он занимался. Я спросил: «Как дела»? Он ответил: «Вообще-то эта идея пришла нам в голову всего две недели назад. Вот, сейчас запускаем в дело». Если вы можете перейти от стадии «Эй, у меня есть идея» к стадии «Давайте запустим эту штуку перед несколькими сотнями серьезных специалистов и посмотрим, как она работает», значит, существует платформа, которая позволяет людям очень быстро опробовать очень интересные вещи. Нельзя сказать, что похожее приложение нельзя было создать пару лет назад, но это потребовало бы гораздо больших затрат.

А при снижении затрат начинает происходить много интересного. Таким образом, первый ответ на вопрос «Почему сейчас?» предельно прост: «Потому что время пришло». Я не могу сказать, почему блоги появились только сейчас; знаю лишь то, что это не имеет ни малейшего отношения к технологии. Абсолютно все технологические аспекты, необходимые для создания блогов, были уже в 1994 году, в тот день, когда был запущен первый браузер Mosaic с поддержкой форм. В нем содержались все необходимые компоненты. Вместо этого появилась служба Geocities. Почему Geocities, почему не блоги? Мы сами не знали, что делали.

Одна идея была плохой; другая, в конечном счете, оказалась хорошей. Нам потребовалось слишком много времени, чтобы понять, что настоящую ценность для людей представляет общение друг с другом, а не простая закачка плохо отсканированных фотографий любимой кошки.

Идея блога появилась около 1996 года с появлением Drudge. Платформы блогов начали появляться, начиная с 1998 года. Настоящее развитие началось около 2000 года.

Лично для меня поворотным моментом стал запуск блога Пеписа Филом Джифордом (Phil Gyford). Дневники Сэмюэля Пеписа, написанные в 1660-х годах, были преобразованы в форму блога, причем каждый день публиковалась новая запись из дневника Пеписа. Насколько я понимаю, Фил имел в виду (и я в это твердо верю), что блоги просуществуют по меньшей мере 10 лет, потому что именно столько времени Пепис вел дневник.

И это был момент проецирования в будущее: у нас появилась инфраструктура, в доступности которой мы можем не сомневаться.

Почему между появлением браузера с поддержкой форм и дневниками Пеписа прошло целых восемь лет? Людям просто нужно время, чтобы привыкнуть к новым идеям и достаточно хорошо понять техническую форму, чтобы найти для нее новые социальные применения.

Другое серьезное изменение заключается в том, что социальное программное обеспечение, создаваемое в наши дни, строится исключительно на базе Web. Если взглянуть на социальные программы в Web в середине 1990-х, многие из них представляли собой специализированные продукты с наспех прилепленным веб-интерфейсом: «Программа Giant Lotus Dreadnought, с Новым Облегченным Веб-Интерфейсом!» Это были не веб-приложения, а неповоротливые махины, украшенные картинками и ссылками, на которых можно было щелкать мышью.

Блог изначально ориентируется на Web-технологии, от начала до конца. Wiki — еще один специфический для Web способ совместной подготовки материалов. Он предельно облегчен, слабо связан, легко расширяется и делится. И это не только поверхностное проявление типа «Смотрите, сколько интересного можно сделать с формами». Wiki предполагает, что в качестве транспорта используется протокол HTTP, а в коде присутствует разметка. RSS — специфический для Web способ объединения информации. Итак, мы берем все эти новые технологии и расширяем их так, чтобы на их основе можно было быстро создать что-то новое.

Третье обстоятельство, ускоряющее развитие социального программного обеспечения, представляет подход к разработке программного обеспечения, выраженный удачной фразой Дэвида Вайнбергера (David Weinberger): «Небольшие, неплотно пригнанные части». Стоит посмотреть, что Джой Ито (Joi Ito) делает в рамках движения Emergent Democracy, даже если само движение вас не интересует. Все началось с того, что однажды во время разговора Ито сказал: «Меня это огорчает. Я сижу здесь в Японии, и знаю, что все эти люди беседуют друг с другом в реальном времени. Я тоже хочу участвовать в групповом общении, поэтому я запускаю конференц-связь.

Но сама по себе конференц-связь — штука неудобная, поэтому я одновременно открываю окно для чата». И тогда, на первом собрании, кажется, Пит Камински (Pete Kaminsky) говорит: «А я открыл Wiki, вот вам URL». И он вводит URL Wiki в окне чата. И люди, участвующие в разговоре, начинают комментировать содержание Wiki, создавать закладки в чате и т. д. Встреча одновременно проходит в трех разных режимах: два в реальном времени (телефонный разговор и чат) и один в режиме комментирования (Wiki).

Вы же знаете, как обычно проходят встречи с конференц-связью: либо один или два человека начинают доминировать, либо все перебивают и обрывают друг друга. Координировать выступления в конференц-связи очень трудно — люди не

видят друг друга, и это усложняет логику передачи права голоса. В конференц-связи Джоя Ито логика передачи переместилась в чат. Участник набирал «Прошу слова», а модератор конференц-связи отвечал ему в чате: «Ты выступаешь следующим». В результате конференц-связь проходила невероятно гладко, потому что чат предоставлял своего рода канал для управления разговором.

Тем временем в чате люди комментировали то, что они слышали в этот момент. «О, это напоминает мне работы Имярек». Или: «Обратите внимание на этот URL... И еще на этот номер ISBN». Обычно в конференц-связи URL приходится диктовать по буквам: «Нет, нет, нет... тройное дубль-вэ, точка, net, дефис...». В окне чата вы видите весь URL и можете сразу щелкнуть на нем, и это позволяет сказать в конференц-связи или в чате: «Открой Wiki и посмотри на это».

Так образуется широкополосная мультимедийная конференц-связь, однако она реализована не в виде одного гигантского монолита. На самом деле это три небольших программных компонента, находящихся поблизости и скрепленных небольшим количеством «социального клея». Эта схема открывает громадные возможности. Она принципиально отличается от подхода «Давайте возьмем махину от Lotus и прилепим к ней веб-интерфейс».

Наконец, четвертой и последней движущей силой современной революции в области социального программного обеспечения является его повсеместное распространение. Web растет уже очень, очень давно. Поначалу лишь узкий круг обладал доступом к Web, потом доступ получили многие люди, а теперь доступ к Web есть почти у каждого. Но сейчас происходит нечто иное. Во многих ситуациях *все* люди обладают доступом к сети. А «все» — это совсем не то же, что «многие». «Все» позволяет считать некоторые вещи само собой разумеющимися.

В наши дни Интернет доступен не во всем мире (и даже не во всех развитых странах). Но некоторые группы людей — студенты, работники современных офисов, высококвалифицированные специалисты — практически всю работу выполняют по сети. Все их друзья подключены к Интернету. Все их домашние подключены к Интернету.

Эта вездесущность начинает восприниматься как нечто само собой разумеющееся. Билл Джой (Bill Joy) однажды сказал: «Мой метод заключается в том, чтобы найти какую-нибудь хорошую мысль и предположить, что она истинна». Постепенно появляются программы, которые просто предполагают, что у любой «отключенной» группы имеется сетевой компонент. В наши дни любая группа, начиная со скаутских организаций, может обладать сетевыми компонентом, легковесным и простым в управлении. И это совсем иное дело, чем старый стереотип «сетевого сообщества». Представьте диаграмму Венна с двумя кругами: левый круг представляет мою реальную жизнь, а правый — мою виртуальную жизнь. Я — то единственное общее, что между ними есть; люди, находящиеся в моем реальном мире, отличны от людей в моем сетевом мире. В течение последних 30 лет это было характерно для Сети: круг друзей в реальной и виртуальной жизни не совпадал. Но если круги на диаграмме сливаются так, что два круга общения совпадают, возникает совсем другая схема. В условиях повсеместного доступа к Сети деление между сетевым и реальным соответствует уже не разным группам, а разным режимам взаимодействия в одной группе.

Также существует вторая разновидность повсеместного распространения сетевых технологий, которой мы пользуемся на этой конференции благодаря сети

WiFi. Если предположить, что группа людей может одновременно общаться и лицом к лицу, и по сети, это открывает новые возможности по сравнению с ситуацией, когда реальное и виртуальное общение рассматриваются как раздельные случаи. Я сейчас не провожу реальных собраний без работающего чата или Wiki. Довольно быстро начинаешь автоматически предполагать, что в группе возможны действия вида: «Да, я прихватил свои слайды PowerPoint, показал собравшимся, а затем скинул в Wiki. Так что теперь все желающие могут загрузить их». Так возникает своего рода общее хранилище групповой памяти. Это принципиально новое явление. Эти виды повсеместного распространения сетевых технологий — и «все подключены к Интернету», и «все, кто находится в комнате, могут находиться в Интернете одновременно» — ведут к возникновению новых схем общения.

Часть 3: Что можно воспринимать как данность?

Если наши предположения истинны (что группа является собственным худшим врагом, и что мы наблюдаем взрывной рост социального программного обеспечения), то что делать дальше? Можно ли сказать что-нибудь определенное о построении социального программного обеспечения, по крайней мере для больших групп с долгим сроком жизни?

Думаю, можно. Немногим более 10 лет назад я уволился с основной работы, потому что сильно заинтересовался Usenet. В то время я думал: «Из этого выйдет нечто грандиозное». Я даже написал книгу «Голоса из Сети» об Интернет-культуре того времени, о Usenet, Well, Echo, IRC и т. д. Книга была опубликована в апреле 1995 года, как раз в то время, когда мир захлестнула волна Web. Но мой исходный интерес не пропал, поэтому я так или иначе рассматривал эту проблему в течение 10 лет, и последние полтора года или около того я занимаюсь ей особенно плотно.

Возникает вопрос: «Что необходимо для обеспечения успешной деятельности крупных долгосрочных сетевых групп?» Пожалуй, теперь я возьмусь ответить на него с некоторой долей уверенности: «Это зависит от обстоятельств». (Надеюсь, в следующие 10 лет мне удастся найти чуть более конкретную формулировку.)

Но я хотя бы могу сказать, от каких обстоятельств это зависит. У кальвинистов существовала доктрина благодати природной и божественной. Природная благодать: «Чтобы попасть на небо, ты должен совершать праведные поступки в миру», а божественная благодать: «...И на тебе должно быть благословение Господне». И вы никогда не знали, обладаете вы божественной благодатью или нет. Так кальвинисты объясняли тот факт, что в Откровении Иоанна Богослова ограничивалось количество людей, которым суждено попасть на небо.

Нечто похожее происходит и с социальным программным обеспечением. Один и тот же фрагмент программного кода может работать во множестве разных сред. Иногда он делает то, что было задумано, а иногда — нет. Так что в группах тоже проявляется нечто сверхъестественное: одного лишь хорошего программного обеспечения недостаточно, потому что социальное поведение групп относится к эффектам, проявляющимся на стадии выполнения. Если заглянуть в Yahoo Groups и проанализировать данные подписки, вы, как и следовало ожидать, обнаружите здесь

экспоненциальный закон. Существует небольшое количество групп с большой численностью, умеренное количество групп с умеренной численностью и длинный хвост групп-неудачников. Неудачи неизменно сопровождают более 50 % списков рассылки в любой категории. Так что рецепта на все случаи жизни не существует. Нельзя сделать что-то такое, чтобы каждая попытка завершалась успехом.

Тем не менее, существует около полудюжины утверждений, справедливых в широком смысле для всех известных групп и всех прочитанных мной правил, обеспечивающих работу крупных долгосрочных групп. И я разобью этот список надвое. Если вы намерены создать социальную программу, предназначенную для поддержки больших групп, вы должны принять три положения (аксиомы) и учесть при проектировании еще четыре.

Три аксиомы, которые необходимо принять

1. Невозможно полностью разделить технические и социальные аспекты. Существуют два заманчивых стереотипных подхода, касающихся пересечения социальных и технологических аспектов. Первый: «Здесь мы будем обрабатывать технологические стороны, а здесь — социальные. Можно создать разные списки рассылки с разными дискуссионными группами; одна дорожка пойдет туда, а другая — сюда». Такой способ не сработает; две стороны неразделимы. Это никогда не было выражено более четко, чем в паре документов, озаглавленных «LambdaMOO идет в новом направлении», уже упоминавшихся ранее. Лучшее, что я могу сделать, — это переадресовать вас к этим документам.

На первый взгляд это кажется очевидным, но это один из тех стереотипов, который бесконечно повторяется на практике. Недавно я участвовал в списке рассылки, посвященном социальному программному обеспечению, и кто-то сказал: «Народ, я знаю! Давайте создадим второй список, в котором будут обсуждаться только технические стороны». Документация LambdaMOO была написана в начале 1990-х, сейчас идет 2003 год, но люди все еще хотят верить в существование четкой границы, отделяющей технологию от поведения обычного пользователя. И что же произошло, когда был создан второй (технический) список рассылки? Ничего. Не произошло ничего. Никто не ушел из первого списка; никто не мог отделить технические аспекты от социальных, потому что разбить уже идущий разговор на две ветки невозможно.

Также существует другой образ мышления, касающийся технической и социальной динамики, который выглядит очень, очень заманчиво: «Действия людей определяются программой!»

И это действительно верно до определенной степени. Но полностью запрограммировать социальные аспекты невозможно — разные списки рассылки работают на базе одних программных пакетов, но обладают совершенно разными культурами. И Slashdot, и Plastic.com работают на одной программной платформе, но их культуры также сильно различаются.

Отделить технологические эффекты от социальных невозможно, как нельзя и задать все социальные аспекты на технологическом уровне. Группа так или иначе

будет отстаивать свое существование независимо от программного обеспечения, и вы столкнетесь с комбинацией социологических и технологических эффектов.

Группа реальна. Она проявляет спонтанно возникающие эффекты. Ее нельзя игнорировать и невозможно запрограммировать. И лучшее решение (или по крайней мере решение, которое срабатывало чаще остальных) состоит в том, чтобы возложить ответственность за определение ценностей и их защиту на саму группу — вместо того, чтобы пытаться определить все заранее в программе.

2. Не все члены группы однородны. В одной из стандартных схем выделяется подгруппа пользователей, которые заботятся о целостности и успехе группы в целом больше остальных. И эти люди становятся основой группы; Арт Клейнер (Art Kleiner) называет их «группа внутри группы, которая имеет наибольшее значение».

Основа группы в Communitree не отличалась от других случайных пользователей, посещавших сервер. В своем внутреннем представлении они были обособлены, потому что хорошо знали, к чему они стремятся, но не могли защититься от других пользователей. Однако во всех известных мне успешных сетевых сообществах возникала основная группа, обеспечивавшая интересы сообщества в целом (а не его отдельной части); фактически она культивировала и улучшала социальную среду, поощряя желательное и осуждая нежелательное поведение.

Если программное обеспечение не всегда позволяет основной группе выразить свои интересы, последняя изобретет новые пути для этого. Так, в alt.folklore.urban (AFU) — группе Usenet, посвященной городскому фольклору — выделилась группа активистов, которые со временем подружились. Многие из них жили в Кремниевой долине, поэтому они решили вместе поехать на барбекю; для координации поездки был создан отдельный список рассылки «Old Hats».

Однако после проведения барбекю список рассылки остался, и членство в нем было предоставлено другим читателям AFU, но только избирательно — тем, кто достаточно долго находился в AFU и был со всеми знаком. Средний читатель AFU даже не знал о существовании списка. Список «Old Hats» стал местом для ведения метадискуссий (то есть дискуссий о деятельности AFU), и его члены начали формально координировать свои действия, если они собирались зафлеймить или игнорировать кого-нибудь в alt.folklore.urban.

С ростом Usenet в AFU появилось много новичков; им понравилась эта группа, потому что ее работа была хорошо скоординирована. Чтобы избавиться от проблем масштабируемости, которые могли бы возникнуть при включении большого количества участников в список «Old Hats», они сказали: «Мы создаем второй список, который будет называться "Young Hats"».

Так в конечном итоге в AFU возникла трехуровневая система, отчасти напоминающая уровни анонимов, зарегистрированных пользователей и людей с высокой кармой на Slashdot. Но поскольку Usenet не позволял основной группе AFU реализовать эту структуру на программном уровне, они ввели новые программные компоненты (эти списки рассылки), необходимые для формирования структуры. Таким образом, вы имеете дело не с однородной массой пользователей — в любой нормальной группе члены, находящиеся на хорошем счету, находят и признают друг друга.

3. Основная группа обладает особыми правами, которые в некоторых ситуациях могут приводить к подавлению прав отдельных личностей. Это положение противоречит либертарианским представлениям, достаточно распространенным в сети, и совершенно очевидно противоречит принципу «один человек — один голос». Но если заменить возможность участия на гражданство, мы нередко встречаем примеры того, когда равноправное голосование неуместно.

В начале 1990-х поступило предложение создать в Usenet группу для обсуждения тибетской культуры, которая должна была называться soc.culture.tibet. И предложение было отклонено при голосовании, главным образом потому, что против него проголосовало множество китайских студентов с доступом к Интернету; они считали, что Тибет — это не страна, а область Китая. И по их мнению, поскольку Тибет не был отдельной страной, места для обсуждения его культуры быть не должно.

Сейчас вполне очевидно, что это решение было неверным. Людям, желающим обсудить культуру Тибета, следовало бы предоставить такое место. Они входили в основную группу. Но поскольку принятая в Usenet модель «один человек — один голос» гласила: «Каждый участник Usenet имеет право голосовать за любую группу», неоднозначно воспринимаемые группы могут быть просто закрыты голосованием.

Представьте, если бы в Соединенных Штатах созданию любой дискуссионной группы против войны в Ираке предшествовал опрос пользователей Интернета (или во Франции при создании групп, выступающей за войну).

Люди, которые хотят вести эти дискуссии, — это те самые активисты, и идея абсолютного равноправия может оказаться вредной, потому что она может привести к тирании большинства. Основной группе необходимы средства защиты, чтобы большая группа оставалась сконцентрированной на своих основных целях и не поддавалась основным инстинктам.

В Wikipedia (сетевая энциклопедия, создаваемая совместными усилиями) существует аналогичная система в виде «добровольческой пожарной дружины» — группы людей, особенно беспокоящихся об успехе Wikipedia. И поскольку такие люди обладают достаточными полномочиями (вследствие особенностей работы Wiki они всегда могут легко отменять граффити и т. д.), вся система продолжает существовать несмотря на многократные атаки. Таким образом, наделение основной группы более широкими полномочиями является действительно мощным средством.

Из-за сложностей с сосредоточением на основной цели у всех сколько-нибудь целостных групп существуют свои своды правил. Неформальные правила существуют всегда, но наряду с ними также иногда существуют правила формальные в виде общедоступного документа. По меньшей мере к формальным правилам можно отнести принципы, воплощенные в программном коде, — «так работает программное обеспечение». Неформальная часть формулируется на уровне «У нас здесь так принято». И независимо от того, что реализовано в программе или записано в местной хартии, неформальная часть будет существовать всегда. Разделить эти две составляющие невозможно.

Даже если вы не примете эти три аксиомы изначально, они все равно проявятся в вашей деятельности. И тогда вам придется писать свой документ типа «Да, мы запустили проект, а потом пришли пользователи и стали делать всякие нехорошие вещи. А теперь мы документируем эту поучительную историю, чтобы будущие поколения не повторяли наших ошибок» — хотя вы так и не прочитали аналогичный документ, написанный еще в 1978 году.

Четыре положения, которые следует учитывать при проектировании

Кроме аксиом, которые придется принять, хотите вы того или нет, также существует ряд факторов, которые должны учитываться разработчиками группового программного обеспечения при проектировании.

1. Первое, что следует учесть при проектировании, — это *идентификаторы*, представляющие пользователей. Я намеренно говорю «идентификаторы», потому что не хочу использовать термин «личность» (identity). Концепция личности в последнее время стала одной из тех идей «с нагрузкой», когда, дернув за нужную веревочку, вы получаете в нагрузку целый мешок ненужного барахла. Тема личности сейчас в моде, но в облегченной модели, необходимой для социального программного обеспечения, вполне достаточно условного идентификатора.

Существует достаточно хорошее понимание того, что анонимность плохо работает в условиях групп, потому что для ведения разговора необходимо как минимум знать, кто, что и когда сказал. Несколько хуже понимается тот факт, что слабая псевдоанонимность тоже работает достаточно плохо, потому что я должен ассоциировать то, что мне говорят сейчас, с прошлыми разговорами.

Лучшая в мире система управления репутацией находится прямо здесь, в нашем мозгу. Почти все разработки в области систем репутации, выполняемые сегодня, либо тривиальны, либо бесполезны, либо и то, и другое сразу, потому что в большинстве обыденных ситуаций для репутации трудно найти явное выражение. Сайт eBay в этом нам помочь не сможет, потому что eBay работает на уровне неповторяющихся атомарных транзакций, не имеющих ничего общего с социальными ситуациями. Система репутации eBay работает на редкость четко, потому что она начинается с простой транзакции («Сколько денег за сколько единиц товара?» и преобразуется в линейную метрику. Такой подход плохо работает в социальных ситуациях, потому что карма (то есть односторонний альтруизм) оказывается куда более тонкой и расплывчатой, чем репутация eBay.

Репутация не обобщается и не переносится. Существуют люди, которые обманывают свою супругу, но не жульничают при игре в карты и/или наоборот. Репутация в одной ситуации не всегда может напрямую переноситься на другие ситуации.

Если вам нужна хорошая система репутации, просто дайте мне запомнить себя. Если вы окажете мне услугу, я это запомню — причем не передним, а задним умом. Когда я в следующий раз получу от вас сообщение, у меня возникнет теплое чувство; я даже не буду помнить, почему. А если вы меня обидите, то при

получении сообщения у меня начнет кровь стучать в висках, хотя я тоже не буду помнить, почему. Если дать пользователям возможность запоминать друг друга, возникает репутация, и все, что для этого нужно, — это простые и в определенной степени устойчивые идентификаторы.

Пользователь должен иметь возможность идентифицировать себя, и смена идентификаторов должна сопровождаться каким-то штрафом. Наказание за смену идентификатора не обязано быть тотальным. Но пользователь, меняющий свой идентификатор в системе, должен потерять часть репутации или контекста. Тем самым обеспечивается функционирование системы.

Конечно, такой подход противоречит настроениям ранних работ по психологии в Интернете. «О, в Интернете все смогут менять личности и пол, как мы меняем носки». Однако ощущение полной свободы в выборе личности разрушается такими историями, как история Кейси Николь.

История довольно долгая, но суть ее проста: одна женщина из Канзаса существовала в сети под видом альтернативной персоны, студентки по имени Кейси Николь. Поскольку друзья выдуманной студентки начали проявлять уж слишком живое участие, женщина решила убить свою сетевую персону и сообщила от имени Кейси Николь, что у нее обнаружено смертельное заболевание.

Привлекательная молодая женщина, с которой все подружились, умирает; что происходит? Все хотят встретиться с ней, пока она еще жива. Женщина запаниковала, и Кейси Николь пропала. В нескольких сообществах Интернета, особенно в сообществе MetaFilter, заподозрили неладное. И десятки людей потратили сотни часов, пытаясь разобраться в происходящем, — своего рода распределенное расследование. В конечном счете, сложив фрагменты из разных посланий Николь, они обнаружили обман.

Теперь многие люди ссылаются на эту историю и говорят: «Вот видите, я же говорил, насколько свободен выбор личности в Интернете!» Но не этот урок следует вынести из истории Кейси Николь; важно другое: смена личности — неординарный шаг. И если сообщество понимает, что вы сделали, это воспринимается как огромный и непростительный проступок. Люди готовы потратить невероятное количество энергии, чтобы найти и наказать вас. Таким образом, личность в Интернете не столь гибка, как нас убеждает ранняя литература; хотя технология упрощает смену личности, социальная жизнь требует определенной степени устойчивости. Все, что вам потребуется, — это система с некой разновидностью постоянных идентификаторов, а уже пользователи наделят их виртуальными личностями и даже эффектами более высокого уровня вроде репутации.

2. Следует разработать механизм, который бы позволял пользователям быть на хорошем счету; некоторый способ признания хороших дел. Как минимум при отображении сообщений должна указываться личность автора. Также возможны более сложные решения, вроде введения формальной кармы, указания срока вхождения в сообществе или особого обозначения привилегированных пользователей, помогающих финансированию системы.

Пока у меня нет полной уверенности относительно того, стоит ли закладывать этот фактор на уровне проектирования, потому что я думаю, что члены группы,

находящиеся на хорошем счету, все равно выделятся среди прочих. Но все больше систем, запускаемых в наши дни, включают дополнительные признаки, по которым можно судить о степени участия членов в системе.

Интересная схема встречается в одной группе обмена музыкой, связывающей Токио и Гонконг. Они работают через список рассылки и пересылают друг другу курьерской почтой FedEx 180-гигабайтные жесткие диски. Пересылаются файлы в формате WAV, а не MP3, притом пересылаются в больших количествах.

Как нетрудно представить, деятельность группы может заинтересовать некоторые организации, не одобряющие подобных вещей. Поэтому при вступлении в группу после имени пользователя указывается имя человека, поручившегося за новичка. Невозможно вступить в группу, пока ваше имя не будет ассоциироваться с именем одного из существующих участников. Эффект репутации возникает от простого связывания двух идентификаторов и начинает действовать немедленно.

В этой системе у пользователей, находящихся на хорошем счету, ссылка на имя поручителя исчезает. С другой стороны, за нарушение интересов группы исключается не только участник, но и его поручитель. Существует великое множество простых способов реализации этой идеи.

3. Участие в группе должно быть ограничено, пусть даже барьеры будут небольшими. Одна из причин, погубивших Usenet, как раз заключалась в отсутствии барьеров при отправке сообщений. Это привело как к общесистемным сбоям вроде спама, так и к намеренным сбоям вроде постоянных атак женоненавистников в группах, посвященных феминизму, или расистских атак в афро-американских группах. Присоединение или участие в группе должно быть сопряжено с некоторыми затратами, если не на самом нижнем, то на более высоком уровне. В этом случае должна быть предусмотрена сегментация возможностей.

Возможен вариант с тотальной сегментацией — либо вы участвуете, либо нет, как в описанной ранее музыкальной группе. Сегментация также может быть частичной; любой желающий может читать Slashdot, анонимы могут отправлять сообщения, не-анонимы могут отправлять сообщения с более высоким рейтингом. Но для модерирования требуется участие в течение определенного времени. По крайней мере, некоторые операции в системе должны быть ограничены, иначе у основной группы не будет средств, необходимых для защиты.

Возможно, это противоречит главной добродетели программирования — простоте использования, однако для социального программного обеспечения такой выбор цели ошибочен: ориентироваться нужно не на индивидуума, а на группу.

Цели групп иногда отличаются от целей отдельных членов, а пользователем социального программного обеспечения является группа, поэтому критерий простоте использования также должен применяться к группе. Если простота использования будет оцениваться только с точки зрения отдельного пользователя, группу будет трудно защитить от атак внутреннего врага — которым, как известно, является сама группа.

Наверное, всем нам доводилось бывать на совещаниях, на которых все отлично проводили время, рассказывали анекдоты и смеялись, и все шло прекрасно...

но до дела так и не доходило. Все развлекались, и цель группы подавлялась личными вмешательствами ее членов.

4. Нужно изыскать способ защитить группу от масштабирования. Большой масштаб убивает общение, потому что для общения требуется интенсивный двусторонний обмен. В контексте общения закон Меткафа — число соединений возрастает пропорционально квадрату числа узлов — не работает. Количество потенциальных двусторонних взаимодействий в группе растет гораздо быстрее размера самой группы, поэтому даже незначительное масштабирование системы приводит к заметному снижению плотности общения. Нужно придумать какой-то способ ограничения круга общения пользователей, чтобы пользователи оставались связанными друг с другом.

Это одно из проявлений проблемы обратной пропорциональности ценности и масштаба. Представьте свой список контактов: тысяча имен, около 150 людей, которых вы считаете друзьями, около 30 близких друзей и 2–3 человека, которым вы готовы пожертвовать почку. Ценность обратно пропорциональна размеру группы. И вы должны придумать механизм защиты группы в контексте этих эффектов.

Иногда проблема решается «мягким ветвлением». Из всех виденных мной сред этот механизм лучше всего реализован в LiveJournal, где концепции «вы» и «ваша группа» тесно переплетены. Средний размер группы LiveJournal составляет около 12 участников, а значение медианы где-то около 5.

Но каждый пользователь немного связан с другими скоплениями через своих друзей, и хотя скопления вполне реальны, они не имеют четких границ — между ними существуют мягкие перекрытия. Это означает, что, хотя многие пользователи участвуют в малых группах, большинство из полумиллиона пользователей LiveJournal связано друг с другом по короткой цепочке.

Некоторые социальные среды (такие, как каналы IRC и списки рассылки) самомодерируются с увеличением масштаба, потому что с ухудшением отношения «сигнал–шум» люди начинают покидать группу, ситуация улучшается, приходят новые люди и т. д. В системе возникают колебания, но в целом производится автокорректировка.

В этой области моим любимым образцом является MetaFilter: при проявлении эффектов масштабирования страница нового пользователя отключается. «Кто-то упоминает нас в прессе и говорит, какие мы замечательные? Пока!» Это способ поднять планку; это создание порога для участия. И каждый, кто ставит закладку на эту страницу и говорит: «Знаете, я действительно хочу здесь быть; возможно, я вернусь позже» — и принадлежит к числу пользователей, желательных для MetaFilter.

Итак, нужно найти механизм защиты пользователей от масштабирования. Впрочем, это не означает, что масштаб системы не должен расти. Но нельзя пытаться увеличивать систему, надувая отдельные разговоры, как воздушный шарик; человеческое общение, построенное по схеме «многие ко многим», надуваться не может. Оно либо рассеивается, либо превращается в широковещательную трансляцию, либо умирает. Так что продумайте заранее, что вы будете делать с масштабированием, потому что это все равно неизбежно произойдет.

Заключение

Конечно, эти четыре фактора являются необходимыми, но не достаточными условиями. Я привожу их скорее как основу для дальнейших интересных экспериментов. Существует великое множество других эффектов, определяющих особые свойства различных аспектов программного обеспечения, поэтому на практике обычно стоит применять сразу несколько распространенных схем. Однако описанные мной архитектурные решения встречаются во многих примерах социального программного обеспечения для крупных долгосрочных групп.

Кроме того, массу интересных возможностей открывает создание подгрупп, будь то гильдии в крупных многопользовательских играх, сообщества в LiveJournal или что-нибудь еще. Можно экспериментировать с артефактами общения, когда участие в группе приводит к созданию некоторых данных. В настоящий момент самый интересный пример артефакта общения встречается в проекте Wikipedia, где продукт является результатом процесса.

Все эти возможности существуют, хотя конечно, они могут изменяться в зависимости от платформы. И все же, я полагаю, я описал «общий стержень» — события, происходящие независимо от того, запланировали вы их или нет, и те принципы, которые желательно планировать при проектировании, потому что на мой взгляд они остаются неизменными в крупных социальных программах.

Разработка социального программного обеспечения — дело сложное. И как я уже говорил, эта работа больше напоминает работу экономиста или политика, нежели программиста. А акт размещения социального программного обеспечения и предоставления доступа к нему ближе к отношениям владельца дома с жильцами, нежели к отношениям владельца склада с хранящимися на нем коробками.

Люди, пользующиеся вашими программами, обладают правами и ведут себя так, как обладатели этих прав — даже если это программное обеспечение принадлежит вам, а они платят за его использование. И если вы начнете игнорировать эти права, то узнаете об этом очень быстро.

Это лишь часть той проблемы, из-за которой теория сообщества Джона Хейгла — «сообщество порождает содержание, а это ведет к коммерции» — никогда не работала. Посмотрите: кто бы ни приходил на форумы Clairol, люди иногда желают поговорить о вещах, не имеющих отношения к продукции Clairol.

«Но мы заплатили за это! Здесь сайт Clairol!» — говорят спонсоры. Неважно. Пользователи приходят сюда друг ради друга. Возможно, они работают на оплаченном вами оборудовании и программах, но пользователи приходят сюда друг ради друга.

Я считаю, что описанные мной закономерности (как аксиомы, так и факторы, учитываемые при проектировании) являются устойчивыми. Просто считайте, что решение этих проблем на социальных платформах является вынужденной мерой, и переходите к построению на этой основе других интересных вещей; я думаю, это и станет настоящем результатом этого периода экспериментирования с социальным программным обеспечением.

Спасибо за внимание.

Клей Ширки

Группа как пользователь: флейм[1] и проектирование социального программного обеспечения

Объявив о необходимости изучения социологии программ, обеспечивающих взаимодействие людей при посредничестве компьютера, Клей Ширки углубляется в изучение специфического феномена флейма. Его работа ассоциируется с трудами великих этнографов прошлого и больше напоминает исследования Маргарет Мид из области сексуальных привычек туземцев Тихого океана, нежели типичную статью о программировании. — *Ред.*

Когда мы слышим слово «программы», большинство из нас представляет себе Word, PowerPoint или Photoshop — короче, инструмент для отдельного пользователя. С точки зрения таких программ компьютер представляет собой замкнутое пространство, самостоятельную среду для выполнения работы пользователем. Большая часть современной литературы и практики в области проектирования программного обеспечения (требования к функциональности, интерфейс, тестирование удобства пользователя) направлена на отдельного пользователя, работающего изолированно от других.

Тем не менее, когда мы проводим опросы по поводу того, чем обычно занимаются пользователи на своих компьютерах, во главе списка обычно оказывается некоторая форма социального взаимодействия — общение, сотрудничество, многопользовательские игры и т. д. Вся практика проектирования программ построена на представлениях о компьютере как о «изолированном ящике», тогда как наше реальное поведение в большей степени ориентируется на представления о компьютере как об «открытой двери», когда устройство используется в качестве входа в социальное пространство.

Мы вполне привыкли к проектированию интерфейсов и взаимодействий между человеком и машиной, но наш социальный инструментарий — программы взаимодействия между людьми, которые в действительности чаще всего применяются пользователями — все еще крайне плохо подходят для решения своих задач.

---

[1] Термином «флейм» обозначается жесткая критика кого-либо в сетевой среде — например, посредством электронной почты или в конференциях. — *Примеч. ред.*

Социальные взаимодействия гораздо более сложны и непредсказуемы, чем взаимодействия человека с компьютером, и эта непредсказуемость становится препятствием для классического проектирования, в центре которого ставится пользователь. В результате программы, ежедневно используемые десятками миллионов людей, либо вовсе игнорируются, либо проектировщики считают, что единственным возможным местом для улучшения является интерфейс взаимодействия пользователя с программой.

Пробел между представлениями «компьютер как изолированный ящик» и «компьютер как дверь» возникает из-за недоразвитой концепции пользователя. Пользователем социального программного обеспечения является не сборище отдельных личностей, а группа. Отдельные пользователи принимают на себя роли, имеющие смысл только в группах: предводитель, последователь, блюститель порядка, распорядитель и т. д. Также существуют различные аспекты поведения, встречающиеся только в группах, от достижения консенсуса до построения карьеры.

И все же, при всех очевидных различиях между личностным и социальным поведением, мы почти не обладаем опытом проектирования, которое бы рассматривало группу как основную сущность, для которой оно производится.

Заполнение этого пробела открывает огромные перспективы и не требует сложных новых инструментов. Все, что потребуется, — это новые подходы к старым проблемам. В самом деле, многие из последних важных работ в области социального программного обеспечения были технически простыми, но социально сложными.

Уроки флеймовых войн

Списки рассылки были первыми общедоступными примерами социального программного обеспечения (система PLATO опередила списки рассылки на десятилетие, но ее пользовательская база была ограничена). Списки рассылки также стали первыми подробно проанализированными виртуальными сообществами. И около 30 лет практически при любом сколько-нибудь подробном описании списков рассылки упоминался флейм — склонность участников списка рассылки пренебрегать стандартами общественных приличий при попытках общения с каким-то невежественным дебилом, который настолько туп, што ни умеет писать без ашибок, и вообще должен СДОХНУТЬ, притом МУЧИТЕЛЬНО, ТВАРЬ ЭДАКАЯ!!!

Но несмотря на 30-летнюю историю описаний, флейм часто рассматривается проектировщиками как побочный эффект, словно все эти извержения аргументов, написанных прописными буквами, кажутся им удивительными или необъяснимыми.

А ведь в флеймовых войнах нет ничего удивительного; это одна из самых устойчивых особенностей в практике списков рассылок. Если оценивать программу по тому, что она делает (а не по целям, заявленным ее проектировщиками), то список рассылки, среди прочего, является инструментом для создания и поддержания горячих споров (впрочем, это относится к любым другим средам общения — WELL, Usenet, форумам и т. д.).

Полярность этих точек зрения: «флеймовые войны как неожиданный побочный эффект» и «флеймовые войны как историческая неизбежность» — обусловлена

двумя основными причинами. Во-первых, средой, в которой работает список рассылки, является компьютер, однако средой для ведения флеймовых войн является людской коллектив. Как бы внимательно вы ни просматривали код программы ведения списков рассылки (допустим, Mailman), вы не найдете в нем комментария: «Следующая функция наращивает взаимное недопонимание между пользователями, что в конечном счете порождает ругань и озлобление». И все же использование программы часто приводит именно к этому результату.

Внутреннее представление пользователя о текстовом редакторе особой роли не играет: если редактор позволяет выводить текст в несколько столбцов, пользователь сможет это сделать, а если нет — то не сможет. С другой стороны, внутреннее представление пользователя о социальном программном обеспечении играет огромную роль. Например, «персональные домашние страницы» и блоги с технической точки зрения чрезвычайно схожи — и то, и другое подразумевает локальное редактирование с глобальным размещением. Различия между ними в основном сводятся к внутренним представлениям пользователя. Понятие блога появилось еще до изобретения самого термина, а термин появился раньше всех современных программ для ведения блогов. Сдвиг произошел во внутреннем представлении пользователя о публикации, а инструментарий последовал за изменениями в социальной практике.

Кроме того, даже если проектировщики программного обеспечения принимают во внимание пользователей, обычно они рассматривают их изолированно. У этой привычки много причин: повсеместный доступ к Сети появился относительно недавно; на концептуальном уровне проще рассматривать пользователей как индивидуумов, а не как исполнителей социальных ролей; исторически практика проектирования ориентирована на индивидуумов, и т. д. В результате на первое место ставится достижение максимальной индивидуальной гибкости, даже если это может создать конфликт с целями группы.

Флейм — этот незапланированный, но неизбежный спутник списков рассылки — стал нашим первым ключом к конфликту между индивидом и группой в опосредованных пространствах. И наша исходная реакция на него также стала первым свидетельством слабости классического проектирования, ориентированного на отдельного пользователя.

Сетевой этикет и kill-файлы

Первой общепринятой реакцией на флейм стало появление сетевого этикета. Сетевой этикет представлял собой рекомендуемый набор правил поведения, предполагавший, что источником флейма являются отдельные пользователи (а кто же еще?). Если объяснить каждому пользователю, что флеймить нехорошо, все перестанут это делать.

В основном эта схема не сработала. Проблема была простой — сетевой этикет был нужен в основном людям, не признававшим его. Эти же люди в наименьшей степени прислушивались к чужому мнению, и их было очень трудно убедить в соблюдении норм сетевого этикета[1].

[1] Кроме того, флейм был их развлечением. — *Примеч. ред.*

Любопытно заметить, что сетевой этикет подошел вплотную к анализу группового феномена. В большинстве версий рекомендовалось, среди прочего, напрямую связываться с флеймерами вместо того, чтобы отвечать им в список. Каждому, кто пытался это сделать, известно, что эта методика оказывалась на удивление эффективной. Но даже здесь коллективные составители сетевого этикета неверно интерпретировали результаты. Прямое обращение к флеймеру работало вовсе не потому, что он осознавал свою неправоту, а потому, что он лишался аудитории. Флейм — не персональное выражение, а своего рода представление, основанное на социальном контексте.

И вот здесь стратегия «прямого контакта» давала сбой. В документации по сетевому этикету прямой контакт обычно описывался как средство, позволяющее обратиться к рациональной стороне флеймера и убедить его отказаться от дальнейшего флейма. Однако на практике среди флеймеров оказывалось очень много рецидивистов. В группе поведение человека изменяется, и хотя обращение «один на один» способно произвести успокаивающий эффект, это изменение происходит в социальном контексте, а не на личном уровне. Когда общение возвращается в условия группы, вместе с ним возвращается и склонность к выступлениям, рассчитанным на внешний эффект.

Другой стандартный ответ на флейм — kill-файлы, иногда также именуемые «фильтрами от идиотов». Такой файл содержит список отправителей, комментарии которых должны отфильтровываться программой до того, как вы их увидите (в фольклоре Usenet даже существует специальный звук *плонк*, издаваемый невоспитанными субъектами при попадании в kill-файл).

Kill-файлы также обычно оказываются малоэффективными, потому что простое удаление одного голоса из флейма почти не способствует улучшению отношения «сигнал/шум»[1] — если флеймеру удастся спровоцировать ответ, удаление только его сообщений не повлияет на волну бессмысленных ответов. И хотя люди постоянно (уже 30 лет) говорят о том, что «если все будут игнорировать X, он сам уйдет», логика коллективных действий делает такой исход почти невозможным — всего пара людей, попавшихся на удочку, провоцируют флеймовую войну, и чем многочисленнее группа, тем сложнее обеспечить соблюдение дисциплины всеми ее членами.

Трагедия общинных земель в общении

Флейм относится к классу экономических проблем, обозначаемых общим термином «трагедии общинных земель»[2]. В двух словах, трагедия общинных земель возникает тогда, когда группа совместно обладает некоторым ресурсом, и у каждого из ее членов возникает стимул к его чрезмерному использованию. (В исходном

[1] Термин «отношение "сигнал/шум" позаимствован из теории обработки сигналов. Высокое отношение «сигнал/шум» говорит о том, что основную часть принимаемых данных занимает сигнал (ценная информация), а посторонние помехи почти отсутствуют. Низкое отношение «сигнал/шум» подразумевает высокий уровень шума. В контексте дискуссионных групп и сетевых сообществ под высоким отношением «сигнал/шум» обычно подразумевается высокая доля интересных обсуждаемых тем. — *Примеч. ред.*

[2] См. http://dieoff.org/page95.htm.

очерке использовался пример с пастухами на общем выгоне. Группа в целом заинтересована в том, чтобы общинные земли как можно дольше оставались пригодными, но каждый индивид был заинтересован в получении максимальной прибыли с общего ресурса, что в конечном счете приводило к «выбиванию» пастбища.)

В случае со списками рассылки (а также другими средами общения) таким общим ресурсом является общественное внимание. Группа в целом заинтересована в поддержании высокого отношения «сигнал/шум» и содержательности общения (даже спорного). Однако отдельный пользователь заинтересован в максимальной возможности выразить свою точку зрения и получении максимальной доли общественного внимания. Это крайне интересное свойство человеческой натуры: «отрицательное» внимание часто приносит людям больше удовлетворения, чем невнимание. И чем больше группа, чем больше вероятность появления участника, действия которого направлены на получение внимания такого рода.

Тем не менее, предлагаемые меры по борьбе с флеймом постоянно уходили от решений, ориентированных на группу, в область решений, ориентированных на отдельную личность. Логика коллективных действий, о которой говорилось выше, значительно снижала эффективность этих персональных решений. А тем временем никаких попыток введения «социальных контрактов» не предпринималось из-за специфического характера «культуры двери» (нежелание уделять внимание социальным аспектам как эффект первого порядка) и ужаса перед цензурой (максимум личной свободы, даже если она противоречит целям группы).

Ответы в блогах и Wiki

При рассмотрении социальных мер защиты от флейма стоит обратить внимание на блоги и Wiki; по количеству флейма эти области даже близко не стоят к традиционным спискам рассылки и другим известным средам. Блоги относительно свободны от флейма из-за малого объема общего ресурса. В экономическом контексте блоги справляются с трагедией общинных земель за счет разделения и приватизации общего пространства.

Каждый аспект мира блогов находится под управлением конкретного блоггера или группы блоггеров, назначающих собственную политику приема комментариев (в том числе и полный запрет на любые комментарии), удаление комментариев от анонимных или недружелюбных посетителей, и т. д. Более того, комментарии почти повсеместно отображаются отдельно от главной страницы, что заметно сужает круг их читателей. Читатели блогов также избавлены от необходимости применять «фильтры от идиотов». Так как характерный для списков рассылки шаблон «все видят всё» никогда не действовал в мире блогов, возможность перехватывать внимание существующих аудиторий практически отсутствует. Даже комментарии, присоединяемые к конкретному сообщению в блоге, обычно видны только для узкого круга читателей самого сообщения.

Wiki, как и блоги, также избегают трагедии общинных земель, но для этого используется полярно иной подход. Если в первом случае все ресурсы находятся в чьем-то ведении, то в Wiki ничего не принадлежит никому. Если список рассылки состоит из отдельных неприкосновенных сообщений с общим пространством

общения, в Wiki даже сам процесс написания имеет совместный характер. Если кто-то захочет выступить в Wiki, материал нарушителя быстро редактируется или удаляется. В самом деле, история Wikipedia, источника совместных материалов по целому ряду неоднозначных тем от ислама до Microsoft, видела многочисленные и в целом неудачные попытки искажения или удаления целых записей. А поскольку старые версии страниц Wiki всегда архивируются, отменить повреждения оказывается проще, чем причинить их (представьте, как выглядели бы наши города, если бы стирать граффити было проще, чем создавать их).

Блоги и Wiki доказывают возможность открытого общения, не страдающего от вмешательства флеймеров, за счет формирования социальной структуры, поощряющей или подавляющей некоторые аспекты поведения. В самом деле, базовый принцип функционирования блогов и Wiki — локальное написание материалов с последующим общим доступом — также является общим принципом функционирования списков рассылки и BBS. В этом свете допущения, используемые программным обеспечением списков рассылки, напоминают не столько Единственно Верный Путь проектирования социального контракта между пользователями, сколько одну стратегию из многих.

Возврат к старому инструментарию

Возможность включения новых социальных компонентов в старые программы открывает невероятные возможности. Классический пример: система модерирования Slashdot передает оценку комментариев в руки самих пользователей. Проектировщики взяли традиционный формат BBS (тематические обсуждения с сортировкой по времени) и включили в него фильтр качества. Но вместо того, чтобы предполагать равенство всех пользователей, проектировщики Slashdot создали систему кармы, которая отдает предпочтение пользователям, склонным к оценке комментариев, полезной для сообщества. А для поддержания порядка в этой системе была создана концепция метамодерирования, решающая проблему «Кто будет стеречь стражника?» (всю информацию можно найти в Slashdot FAQ[1]).

Рейтинги, карма, метамодерирование — каждая из этих систем относительно проста в технологическом отношении. Однако их применение дало возможность Slashdot поддерживать огромную пользовательскую базу, с поощрением участников, публикующих интересные материалы, и ограничением оскорбительных или неуместных сообщений.

Аналогично, на сайте Craigslist за основу был взят список рассылки, в который были включены некоторые простые функции с глубокими социальными эффектами. Поскольку Крейг Ньюмарк имел коммерческие стимулы к тому, чтобы его сайт успешно работал, он со своими коллегами удалял сообщения, помеченные как «неподходящие» достаточным количеством читателей. Как и Slashdot, он нарушает исходную предпосылку о том, что социальное программное обеспечение не должно устанавливать групповых ограничений для индивидуального участия,

[1] См. http://slashdot.org/faq/com-mod.shtml#cm600.

причем благодаря этому обстоятельству Craigslist лучше работает. С положительной стороны, включение кнопки номинации «лучшего материала Craigslist» в каждое сообщение создает социальный стимул для публикации интересных материалов. Кнопка «лучшего материала» идеально демонстрирует слабость ориентации на отдельного пользователя. Если бы среда Craigslist оптимизировалась для отдельного пользователя, наличие такой кнопки выглядело бы нелогично — если вам понравилось конкретное сообщение, вы просто сохраняете его на своем жестком диске. Но пользователи не ограничиваются сохранением сообщений; они щелкают на этой кнопке. Как и в случае с флеймом, кнопка предполагает действия пользователя, рассчитанные на реакцию аудитории, но здесь этот механизм обращен во благо. Сообщение номинируется на звание «лучшего материала» по единственной причине: вы хотите, чтобы другие увидели его — иначе говоря, если вы действуете в групповом контексте.

Новое отношение к социальной проблематике

Bumplist[1] Джона Брукера-Коэна (Jonah Brucker-Cohen) заслуживает внимания как эксперимент с социальными аспектами списков рассылки. Bumplist с его девизом «почтовое сообщество для непреклонных» представляет собой список рассылки для шести человек, к которому может присоединиться любой желающий. Когда к списку присоединяется седьмой пользователь, то первый автоматически исключается из него, а если он захочет вернуться, то присоединяется заново, исключая второго пользователя, и так далее до бесконечности (на момент написания статьи список Bumplist включал 87 414 подписчиков и 81 796 повторных подписчиков). Bumplist преследует скорее полемические, нежели практические цели: Брукер-Коэн описывает его как попытку переосмысления культуры и правил списков рассылки. Тем не менее, Bumplist наглядно демонстрирует, как простые изменения в хорошо известных средах могут порождать принципиально новые социальные эффекты.

Можно легко представить себе множество аналогичных экспериментов. Например, что нужно сделать для создания списка рассылки с торможением флейма? Стоит перестать рассматривать всех пользователей как изолированных индивидов, как открывается масса возможностей. Можно ввести искусственную задержку — скажем, если пользователь создает пять сообщений в течение часа, к каждому последующему сообщению будет добавляться накапливаемая 10-минутная задержка. В конечном счете каждое сообщение будет доставлено, но задержка предотвратит скоропалительные ответы, главную основу флеймовых войн, и создаст «период охлаждения» для самых пылких участников. Можно изобрести своего рода «отстойник для тем», включая в каждое сообщение «кнопку осуждения» в стиле Craiglist. Бесконечные, бессмысленные темы (например, «Какая Операционная Система Объективно Является Лучшей?») отправляются в отстойник, если против них проголосовало достаточное количество пользователей. (Хотя

[1] См. http://www.runmc.org/feature/read/+BumpList/+86/.

пользователи могут изменять заголовки тем и обходить это ограничение, существует один интересный момент, впервые подмеченный Джулианом Диббелом (Julian Dibbell): пользователи часто уважают отрицательное мнение сообщества, даже не уважая отрицательного мнения отдельных участников[1]).

Можно ввести функцию «автоматической изоляции»: любой диалог, в котором только два пользователя перекидываются шестью или более сообщениями (подставьте любое число по своему вкусу), автоматически оформляется в подсписок, ограниченный этой парой. Материал будет архивироваться и останется доступным для заинтересованных читателей, но разговор будет продолжаться без отвлечения внимания основной аудитории.

Нетрудно представить себе аналогичную функцию, работающую на более общем уровне отношения «сигнал/шум» и основанную на том факте, что в списке рассылки всегда имеется самый активный участник, который отправляет сообщения гораздо чаще даже второго по активности, и намного, намного чаще участника среднестатистического. Как ни странно, самый активный участник часто даже не подозревает о том, что он занимает эту позицию (в виртуальных сообществах тоже трудно взглянуть на себя со стороны), но если сообщить ему об этом, он часто начинает действовать более сдержанно. Скажем, можно ввести особую пометку всех сообщений от самого активного участника, кем бы он ни был в данный момент, или ограничить максимальное количество сообщений от любого участника неким числом, кратным среднему.

И так далее. Количество возможностей для экспериментов огромно, причем такие возможности существуют в любом социальном контексте, а не только в средах общения.

Быстрые итеративные эксперименты

Хотя большинство таких экспериментов особой пользы не принесет, для выявления позитивного влияния лучше всего воспользоваться быстрыми итеративными экспериментами. Из Slashdot FAQ четко видно, что установившаяся система «рейтинги + карма + метамодерирование» могла появиться только в результате постоянной регулировки. Это стало возможным благодаря относительной простоте технических аспектов и быстрой обратной связи с пользователями.

И все же подобные эксперименты пока остаются скорее исключением, чем правилом. За 30 лет основная техническая работа по ведению списков рассылки была в основном административной — та же программа Mailman предоставляет в распоряжение администратора списка более 100 регулируемых параметров, многие из которых могут принимать несколько значений. Тем не менее, социальные аспекты списков рассылки за эти три десятилетия почти не изменились. И дело даже не в технологической сложности экспериментов с социальной стороной — просто она кажется концептуально чуждой. Представление о компьютере как о «изолированном ящике», используемом отдельным индивидом, стало настолько

[1] См. «Rape in Cyberspace» (http://www.juliandibbell.com/texts/bungle.html). Поищите по словам «aggressively antisocial vibes».

всепроникающим, что многие привыкли к нему. Они тратят время на совершенствование любых аспектов списков рассылки, кроме социальных взаимодействий, для которых эти списки, собственно, и предназначались.

Но если рассматривать групповое мышление как часть среды, в которой работает программное обеспечение, перед экспериментатором открывается целая Вселенная новых возможностей. Социальная сторона даже относительно старых сред — таких, как списки рассылки — содержит множество неопробованных моделей. Конечно, нет никаких гарантий, что любой эксперимент окажется эффективным. Циклы обратной связи в области социальной жизни всегда приводят к непредсказуемым результатам. Все поклонники идей социального совершенствования или тотального контроля будут жестоко разочарованы, потому что пользователи регулярно отвергают любые попытки повлиять на их поведение — либо используя недочеты системы, либо покидая ее.

Но благодаря широте возможностей и простоте потенциальных экспериментов, простоте получения обратной связи от пользователей, а самое важное — благодаря важности социального программного обеспечения в глазах пользователей, даже относительно малое число успешных усовершенствований, при всей их простоте и итеративности, могут привести к непропорционально успешному результату. Так было с Craiglist и Slashdot, и, несомненно, так же будет и с другими экспериментами такого рода.

Эрик Синк

Заполнение промежутка, часть 1

> Я начал писать длинное предисловие к этой статье, а потом передумал — я настолько глубоко согласен со всем, что говорит Эрик Синк, что в долгом предисловии нет смысла. Все, что говорит Эрик Синк, правильно. Поступайте так, как он рекомендует, и вы не ошибетесь. — *Ред.*

В какой-то момент деятельности мелких независимых поставщиков (НП) программного обеспечения клиент обменивает деньги на программы. Ни одна рубрика, посвященная коммерческим аспектам программирования, не может считаться полной без обсуждения этого волшебного события.

Для начала мне хотелось бы определиться с терминологией. До того, как клиент сделал покупку, я обычно говорю о наличии «промежутка». Этот промежуток представляет собой расстояние между потенциальным покупателем и вашим продуктом, и выглядит примерно так:

Чтобы продажа состоялась, этот промежуток должен быть заполнен. Пока этого не произойдет, он представляет всю совокупность проблем и препятствий, мешающих клиенту сделать покупку:

- Клиент никогда не слышал о вашем продукте.
- Клиент не обладает достаточной информацией о вашем продукте.
- Ваш продукт стоит слишком дорого.
- Для оформления покупки клиент должен получить согласие на двух уровнях руководства.
- В продукте отсутствуют функции, необходимые клиенту.
- Продукт не взаимодействует с другими программами клиента.
- Продукт еще не успел «созреть» и не отвечает ожиданиям клиента.

Чтобы ваша фирма продолжила свое существование как коммерческая организация, она должна снова и снова находить способы заполнения этого промежутка. Существуют ровно два возможных метода:

1. Смещение продукта вправо. Расскажите миру о своем продукте. Улучшите свой продукт, чтобы люди захотели его приобрести.
2. Смещение клиента влево. Найдите потенциальных покупателей. Убедите их купить ваш продукт.

Начнем со второго метода — как подвести клиента к вашему продукту?

Активная продажа

Давайте определим еще несколько терминов. Слово «продажа» отчасти перегружено всевозможными смыслами, но в контексте профессиональной функции я предпочитаю обозначать им процесс активного поиска новых потенциальных клиентов и индивидуальную работу с ними с целью убедить их сделать покупку. Человек, который выполняет эту функцию, называется продавцом[1].

Чтобы объяснение стало более четким, я должен объяснить пару вещей, которые не относятся к понятию «продажи».

Маркетинг к продаже не относится. Маркетинг — совершенно отдельная область. К нему относится множество различных операций, от планирования общей стратегии до общения. К продажам все это отношения не имеет. Продавец является клиентом группы маркетинга. Маркетинговые операции обычно основаны на связях типа «один ко многим», тогда как продажа почти всегда базируется на связях типа «один к одному». Продажа связана исключительно с завершением сделки. Сотрудник вашего офиса, занимающийся обработкой поступающих заказов от клиентов, не является продавцом. Да, он имеет дело с «продажей» в бухгалтерском смысле, но я называю эту профессиональную функцию «обслуживанием клиентов».

Определяющей характеристикой продавца является способ оплаты его услуг. Его общее вознаграждение включает комиссионные. Продавец получает определенный процент с каждой успешной сделки. Процент сильно зависит от всевозможных факторов, и я не буду приводить даже приблизительные цифры. Тем не менее, я хочу отметить, что комиссионные часто составляют большую сумму. Во многих организациях самым высокооплачиваемым работником является продавец, который нередко зарабатывает больше генерального директора.

Работа с продавцом

Люди, выполняющие другие профессиональные функции, часто недолюбливают продавцов из-за их комиссионных. Например, главный проектировщик может резонно поинтересоваться, почему продавец получает долю с каждой продажи,

---

[1] Объяснения по поводу неполиткорректного использования мужского рода (почему не «продавщица») убираю — к счастью, этот идиотизм до нас еще не дошел. — *Примеч. перев.*

а он — нет? Неужели продавец более ценен чем тот человек, который создал продукт?

Хорошие продавцы понимают эту проблему и стараются бороться с ней, «подлизываясь» к разработчикам при каждой возможности. Например, если продавцы и разработчики сидят вместе в баре, даже не обсуждается, кто платит за напитки.

С другой стороны, нельзя позволять продавцу проводить слишком много времени с разработчиками. Без должных ограничений продавец начнет приглашать разработчиков при каждом посещении клиента. Внезапно высокооплачиваемый программист превращается в высокооплачиваемого помощника продавца. Типичное решение проблемы — размещение продавцов в отдельном офисе, желательно в другом городе. Продавцы все равно должны находиться поблизости от крупного аэропорта.

Характеристики продавца

Продавец должен обладать особыми качествами. Как правило, это экстраверты, обладающие превосходными навыками межличностного общения. Они весьма самоуверенны, иногда чрезмерно. Как правило, они хорошо выглядят и щегольски одеваются. Продавцы знают о технологии достаточно, чтобы быть опасными. Они умеют с честью выходить из многих непредвиденных ситуаций. Они точно знают, в какой момент следует двинуться к завершению переговоров и предложить сделку.

Между стереотипами продавцов существуют допустимые отклонения, но одно требование является абсолютно обязательным: единственным побудительным мотивом хорошего продавца являются деньги. Одна из самых опасных ошибок при подборе персонала — нанять продавца, который думает о чем-то еще, кроме денег.

Хороший продавец делает только то, что обеспечивается финансовыми стимулами (комиссионными). Все остальное является напрасной тратой времени, и продавец фактически невосприимчив к любым другим методам воздействия.

Помните, что у каждой медали есть оборотная сторона. Хотя я считаю, что «природа наемника» является важным качеством продавца, но наряду с достоинствами есть и недостатки. Хороший продавец прекрасно умеет слушать, но это умение, похоже, действует только тогда, когда он работает над сделкой. Слушая потенциального покупателя, он улавливает и понимает каждую отдельную деталь, никогда не упуская ничего, что приблизило бы к завершению сделки. С другой стороны, представим, что в кабинет входит начальник продавца и говорит: «Фред, нам надо поговорить. До меня дошли слухи, что нашим разработчикам пришлось самим покупать пиво в баре вчера вечером, хотя ты там был. В чем проблема? Это пиво обошлось бы тебе лишь в несколько долларов. При тех деньгах, что ты зарабатываешь, ты вполне мог бы заплатить по их счетам».

Действительно выдающийся продавец услышит из всего сказанного следующее:

«Бу, бу бу бу. Бу бу бу бу бу бу бу бу бу ПОКУПАТЬ бу бу бу бу бу, бу бу бу бу. Бу бу бу? Бу бу бу бу бу бу бу бу ДОЛЛАРОВ. Бу бу ДЕНЬГАХ, бу бу бу, бу бу бу бу ЗАПЛАТИТЬ бу бу бу».

Итак, хотя ген жадности иногда может создавать проблемы, преимущества все же определенно перевешивают недостатки. Управление продавцом, думающим только о деньгах, оказывается делом простым, и в целом обходится без неожидан-

ностей. Вам не нужно подолгу разбираться в мотивации ваших продавцов, как с разработчиками. Все, что требуется, — позаботиться о том, чтобы комиссионные четко соотносились с тем объемом работы, который он должен выполнять.

У всего сказанного имеется логическое следствие: если продавец делает что-то незапланированное, есть только два возможных объяснения:

- Это плохой продавец, потому что он думает о чем-то, кроме денег.
- Виноваты вы сами, потому что продавец просто делает то, что вы «стимулировали» его делать.

Например, предположим, что продавец быстро заключает сделки с клиентами, но очень многие клиенты проявляют недовольство вскоре после покупки продукта. Обычно это бывает в том случае, если продавец «впаривает» продукт людям, которым он на самом деле не нужен. К тому моменту, когда клиент это осознает, продавец уже получил свои комиссионные и переключился на другую жертву. Вы забыли включить довольство клиента в задачу продавца, поэтому он и не тратит лишнего времени. К счастью, у этой конкретной проблемы есть довольно простое решение. Например, основная часть комиссионных остается у вас, пока вы не убедитесь, что клиент остается довольным 90 дней после покупки.

Если продавец тратит слишком много времени с разработчиками, начните штрафовать его за это. Вычитайте $100 за каждый час, проведенный с программистами. Справедливости ради предоставьте ему несколько свободных часов, потому что для заключения сделок продавцу потребуется техническая информация[1].

Еще одно обязательное качество продавца

Любой продавец, который обидится на злые шуточки в этой статье, вероятно, не компетентен.

Продавец просто обязан быть толстокожим. Если его задевает моя карикатура, то что он будет делать, когда клиент ему скажет «нет»? Даже лучшему продавцу чаще приходится слышать «нет», чем «да». Если продавец не может принять отказ без переживаний, вероятно, он занимается не своим делом.

Когда нужен продавец

Хороший продавец — весьма ценный ресурс. Да, ему приходится платить уйму денег в комиссионных, но он по определению приносит еще больше. Решение о найме продавца должно приниматься крайне взвешенно. Принимая на работу продавца, вы навсегда измените свою компанию. Не делайте этого, если нет крайней необходимости. Приведу лишь несколько стандартных причин.

---

[1] Возможно, вы попадете в бесконечный цикл «подгонки стимулов». На каждую введенную вами поправку продавец будет находить обходные пути, вы будете исправлять их, и т.д. Не переживайте; это фундаментальная проблема человеческой натуры. — *Примеч. ред.*

Причина № 1: продукт не пользуется спросом

Из-за специфики некоторых продуктов их продажа требует гораздо больших усилий. Некоторые продукты попросту малоинтересны. Например, никто не хочет покупать страховку жизни. Да, люди покупают страховку жизни, но не с таким энтузиазмом, как новый фильм, только что вышедший на DVD. Вот почему в Америке нельзя размахнуться клюшкой для гольфа, не задев продавца страховок. Без этих людей количество продаваемых страховок было бы гораздо меньше.

Кроме того, некоторые продукты предназначены для людей, которые еще не осознают, что у них имеется проблема.

Если продукт решает проблему, которая не особенно беспокоит клиента, продавец поможет понять потенциальным покупателям, до чего им на самом деле плохо живется. Эта тактика называется «созданием недовольства текущим положением».

Если же вы только начинаете свою карьеру в новой фирме-разработчике, лучше избежать этой ошибки. Начните с минимального промежутка. Выберите продукт, который будет производить впечатление на людей.

Причина № 2: продукт очень дорог

Более дорогие продукты гораздо чаще требуют привлечения продавца, помогающего клиенту принять решение. Например, в покупке дома или машины обычно (хотя и не всегда) используется продавец. Аналогичная проблема возникает и с дорогими программными продуктами. Большие заказы сопряжены с большими решениями, и многие организации принимают решения лишь после долгих колебаний.

Причина № 3: продукт перестал совершенствоваться

По мере того, как программный продукт «зреет» со временем, над ним все чаще работают продавцы, а не разработчики. Если же жизненный цикл продукта близок к завершению, часто попадаются компании с большим количеством продавцов и без единого разработчика. Причина интуитивно понятна: продукт перестает двигаться по направлению к клиенту. Чтобы заполнить промежуток, приходится постоянно тащить клиента к продукту.

(Мне очень хотелось бы привести несколько примеров, но мои редакторы в Microsoft потратили массу сил, обучая меня хорошим манерам. Пусть они думают, что их усилия не пропали даром).

Вариант «Продавец не нужен»

Многие НП прекрасно обходятся вообще без продавцов. Я понимаю, что этот вариант будет считаться ересью в ортодоксальной церкви бизнеса, но я настаиваю на нем. Впрочем, проповедники этой религии и так меня недолюбливают. Если я когда-нибудь узнаю, где находится их собор, я надену очки в стиле Гручо Маркса[1] и спляшу на алтаре.

---

[1] Американский комик. — *Примеч. перев.*

Как обычно, я охотно признаю, что из этого правила есть свои исключения. Тем не менее, слишком многие компании начинают искать продавца до того, как это действительно следует делать.

Перемещение клиентов — очень сложная задача, особенно для небольшой компании с минимальными возможностями воздействия. Клиенты тяжелы и инертны. Они не желают никуда перемещаться и часто обижаются, когда кто-нибудь пытается это сделать. Компания запросто может потратить огромные усилия, пытаясь подвести клиента к продукту, но так и не справиться с сокращением промежутка.

Все эти усилия лучше направить в другую сторону. Усовершенствуйте свой продукт. Переместить продукт к клиенту гораздо проще. Прислушивайтесь к своим клиентам и давайте им то, что они хотят. Пусть ваши клиенты останутся довольными (и расскажут всем своим друзьям, какая у вас замечательная компания).

Концентрация усилий по сокращению промежутка со стороны продукта обладает двумя очень приятными свойствами:

- Она лежит полностью в пределах вашей компетенции. Вы сами знаете, как сделать свой продукт лучше. Вернее, если вы не знаете, как сделать свой продукт лучше, ваша фирма все равно разорится, так что присутствие продавца вас не спасет.
- Улучшая свой продукт для одного клиента, вы улучшаете его и для всех остальных.

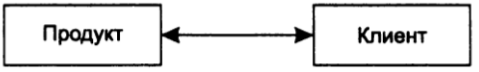
Резюме

Рано или поздно кто-нибудь скажет, что ваша компания — не настоящая, потому что у вас нет продавцов. Ерунда. Не нанимайте продавцов, пока без них можно обойтись.

Эрик Синк

Заполнение промежутка, часть 2

В прошлом месяце мы рассмотрели концепцию, которую я назвал «промежутком»:



Промежуток представляет расстояние между потенциальным покупателем и вашим продуктом. Пока промежуток продолжает существовать, у клиента меньше программ, а у вас меньше денег. Чтобы продажа состоялась, промежуток должен быть заполнен. Пока этого не произойдет, он представляет всю совокупность проблем и препятствий, мешающих клиенту сделать покупку.

Вы как Главный Специалист по Продажам в своей фирме должны решить, каким образом будет заполняться промежуток. Существуют ровно два метода:

1. Смещение продукта вправо.
2. Смещение клиента влево.

В прошлый раз мы говорили об «активной стратегии продаж», или «смещении клиента влево». Сейчас речь пойдет о другом способе заполнения промежутка: смещении продукта вправо.

Пассивная продажа

В первой части этого очерка я утверждал, что большинству мелких независимых производителей программного обеспечения продавцы не нужны, и такие фирмы не должны использовать стратегию активных продаж. В этом месяце мы рассмотрим альтернативный подход и поговорим о *пассивных* продажах. Для начала стоит выделить различия между этими двумя моделями:

- В модели активных продаж ведущая роль отводится продавцу. Он инициирует контакты с потенциальными покупателями, рассказывает о продукте, отвечает на вопросы, остается на связи. В конечном счете, он же убеждает клиента совершить покупку, получает деньги с клиента и доставляет продукт.

- В модели пассивных продаж ведущая роль принадлежит клиенту. Он связывается с вашей компанией только тогда и только в том случае, если он того желает. Он узнал о вашем продукте от своих знакомых, из рекламы или из блога. Он собрал всю доступную информацию о продукте и его возможностях, а теперь связывается с вашей компанией, чтобы задать дополнительные вопросы. Клиент совершает все действия в том темпе, в котором это удобно ему и его организации. В конечном счете он принимает решение о покупке, связывается с вашей компанией и обменивает деньги на продукт.

Возможно, из-за контраста между этими описаниями идея пассивных продаж покажется вам малопривлекательной. В конце концов, стоит ли доверять клиенту все эти важные задачи?

Да, стоит.

Я согласен, что идея пассивных продаж на первый взгляд пугает. Все выглядит так, словно вы доверяете критически важный проект исполнителю, которого вы не знаете, а может быть, никогда не встречали.

Но этот риск вознаграждается. Дело в том, что клиентам нравится, когда им доверяют. Им нравится принимать решения без давления со стороны продавца. Им нравится быть главными.

По всем указанным причинам модель пассивных продаж отлично работает, пока вы соблюдаете свою часть сделки — реакцию на потребности клиента. Да, вы разрешаете клиенту проявлять инициативу, но это не значит, что вы бессильны.

Более того, вам придется основательно потрудиться. Ваша задача — по возможности упростить весь процесс для клиентов. Они будут сами принимать решение о заполнении промежутка. Вы всего лишь сдвинете свой продукт вправо, чтобы промежуток было проще заполнять.

Чтобы добиться успеха в пассивных продажах, необходимо соблюдать семь условий.

1. Позаботьтесь о том, чтобы клиенты знали о вашем продукте

Клиент не купит ваш продукт, если он никогда не слышал о нем. Если кому-то это обстоятельство покажется чем-то новым и оригинальным, вероятно, для него будет таким же откровением узнать, что небо голубое.

Серьезно, я хорошо понимаю, что говорю очевидные вещи, но наличие информации о вашем продукте является важным предварительным условием, особенно для пассивных продаж. Если клиент никогда не свяжется с вами, вы не сможете отреагировать на его потребности. Если вы не знаете, как сообщить людям о существовании своего продукта, наверное, остаток статьи читать не обязательно. Модель пассивных продаж не будет работать, пока продукт не начнет пользоваться определенной известностью. Конечно, следует напомнить, что распространение информации — задача из области маркетинга, а не продаж. Точнее, это составная часть подкатегории, называемой «маркетинговыми коммуникациями». Полное рассмотрение темы маркетинговых коммуникаций выходит за рамки статьи, поэтому я ограничусь тремя краткими рекомендациями.

Будьте внимательны с рекламой

ВОПРОС: в чем различия между покупкой рекламы в журналах и сжиганием долларовых купюр?

ОТВЕТ: Сжигание денег может принести пользу из-за выделения тепла.

Эта шутка взята из статьи о рекламе мелких фирм-разработчиков[1], написанной мной в прошлом году. Далее в статье я говорю о том, что я шучу лишь наполовину. Реклама — дело пугающее и рискованное. Вы можете потратить огромные деньги, совершенно не представляя, зачем это было сделано.

Я не хочу сказать, что мелкие фирмы не должны давать рекламу. Скорее, я говорю, что вы должны быть очень осторожны. Если у вас есть какие-то сомнения, просто подождите. Скажите агенту по рекламе, чтобы он перезвонил вам через полгода.

Используйте специализированные выставки

Среди традиционных мероприятий из области маркетинговых коммуникаций я больше всего люблю специализированные выставки-ярмарки. Помните старую шутку про пиццу и секс? Выставки относятся к этой же категории: даже когда они плохи, все равно они достаточно хороши. Даже на самой худшей выставке, которую я когда-либо видел, я узнал кое-что новое и повстречал интересных людей. Если в вашем сегменте рынка проводятся хорошие выставки, подумайте о том, чтобы принять в них участие.

Ведите разработки «в открытую»

Традиционные методы маркетинговых коммуникаций занимают свою нишу, но, наряду с ними, существуют и новые подходы. В наши дни Интернет стал настолько вездесущим, что одним из лучших способов распространения информации о продукте стала его разработка «в открытую». Иначе говоря, используя блоги, открытые обсуждения и загрузку демонстрационных версий, дайте возможность потенциальным клиентам следить за тем, как идет работа, и общаться с вами. Представьте себя в роли шеф-повара в китайском ресторане: клиенты наблюдают за тем, как вы жарите в кипящем масле креветок и овощи.

Начните с блога — открытого журнала с информацией о ходе разработки. Время от времени публикуйте обновления о том, как идет работа над приложением. В определенный момент приложение будет готово для построения демонстрационной версии для потенциальных клиентов. Опубликуйте общедоступную демоверсию. Организуйте список рассылки или веб-форум для получения обратной связи.

Разработка программного обеспечения требует времени. Ведение работ «в открытую» позволяет эффективно использовать это время для одновременного распространения информации.

---

[1] См. http://software.ericsink.com/Magazine_Advertising.html.

2. Убедитесь в том, что ваш продукт нужен клиентам

Простите, я снова повторяю очевидные вещи, но факт остается фактом: если продаваемый продукт никому не нужен, промежуток становится просто огромным.

Хороший продавец в активной модели способен решить эту проблему. Тактика продажи вещей, которые никому не нужны, отлично известны. Кто из нас стал бы покупать средство для защиты от ржавчины для новой машины, если бы его пришлось специально разыскивать?

В модели пассивных продаж у вас практически нет надежды продать продукт, непривлекательный для круга потенциальных покупателей. Таким образом, очень важно заранее подготовиться и убедить себя в том, что создаваемый вами продукт должен быть востребован. Это другая половина маркетинга.

Проведите позиционирование

Каждый, кто читал хотя бы одну книгу о классическом маркетинге, наверняка встречал слово «позиционирование». В сущности, позиционированием[1] называется процесс определения того, как продукт будет воспринят рынком. С чем должен ассоциироваться этот продукт? С какими другими продуктами он должен сравниваться? Ответ на эти вопросы станет важным шагом, гарантирующим востребованность вашего продукта.

Выберите конкуренцию

Вероятно, попытка избежать конкуренции — самый надежный путь к созданию продукта, который оказывается никому не нужен. Вам нужна конкуренция[2]. Избегая конкуренции, вы одновременно избегаете клиентов. Концепция вашего продукта проверяется наличием других фирм-разработчиков, с выгодой продающих нечто подобное. Если на рынке у вашего продукта нет ни одного аналога — это тревожный признак.

Ведите разработки «в открытую»

Кажется, это правило вы уже где-то видели, не правда ли?

Да, я уже приводил аргументы в пользу ведения разработки «в открытую», но сейчас я привожу их снова по другой причине. Разработка «в открытую» — не только хороший способ распространения информации. Она также позволяет судить о том, насколько людям интересен данный продукт.

Допустим, вы выбрали разработку «в открытую» с публикацией больших объемов информации и предварительных версий на очень ранней стадии цикла разработки. Вы публикуете соответствующие объявления в конференциях и на форумах. Тем не менее, количество загрузок минимально, в список рассылки никто не пишет, и вы не получаете никакой обратной связи.

Плохие новости: похоже, разрабатываемое вами приложение никому не нужно. Хорошие новости: разработка «в открытую» позволила узнать вас относительно

---

[1] См. http://software.ericsink.com/Positioning.html.

[2] См. http://software.ericsink.com/Choose_Your_Competition.html.

рано. У вас еще осталось время, чтобы изменить функциональность приложения. А может быть, вы предпочтете свести риск к минимуму и похороните проект. Так или иначе, плохие новости лучше узнавать как можно раньше, до того, как работа над приложением будет завершена.

3. Убедитесь в том, что продукт не слишком дорог

Цена продукта влияет на размер промежутка.

Среди авторов, пишущих на темы ценообразования, модно писать, что продукт должен быть дорогим, а не дешевым. Основная идея состоит в том, что выбор цены подразумевает некое заявление. Устанавливая высокую цену, вы сообщаете миру, что по вашему мнению, это очень ценный продукт. В результате это должно способствовать повышению спроса.

Некоторые покупатели действительно предпочитают покупать более дорогие продукты. В качестве примера могу привести самого себя. В настоящее время я тренируюсь в спортивной ходьбе на полумарафонскую дистанцию. Для этого очень важно иметь хорошую обувь. Вероятно, мне следовало бы пойти в какой-нибудь из новомодных магазинов, в которых продавец анализирует видеосъемку вашей походки и помогает подобрать идеальные кроссовки. Но я вечно тороплюсь и предпочитаю более простые решения. Я покупаю кроссовки только в том случае, если они выпущены известной фирмой и стоят не менее $ 85 за пару. Этот способ примитивен, но он прост и подходит для меня.

Некоторые клиенты покупают программы по тому же принципу, по которому я покупаю кроссовки. Приобретение самого дорогого продукта удобно для покупателя, у которого нет времени на подробные исследования.

Высокая цена может оказаться привлекательной для клиентов, ищущих либо престижа, либо исключительного качества. Тем не менее, у низкой цены тоже есть свои преимущества. Дело в том, что у многих потенциальных клиентов имеется бюджет. Если цена превышает предельную, промежуток может стать бесконечно большим.

Несколько месяцев назад моя компания снизила цену на нашу систему контроля версий (SourceGear Vault). Даже за исходную цену программа была одной из самых дешевых в своем сегменте рынка. С новой ценой все конкурирующие продукты стоили в несколько раз дороже нашей. Мы знали, что это большой риск. Некоторые клиенты автоматически предполагают, что если продукт конкурентов стоит в семь раз дороже, то он наверняка в семь раз лучше нашего.

Пока риск окупается. Мы приняли это решение, потому что полагали, что промежуток слишком широк для многих покупателей. Похоже, оценка была правильной. С момента изменения цены наши суммарные доходы существенно возросли.

4. Предоставляйте полнофункциональные демо-версии

Каждая фирма-разработчик должна предоставлять своим клиентам возможность опробовать программный продукт перед его приобретением. Не делать этого просто абсурдно. Клиенты приходят на ваш веб-сайт, ожидая найти на нем демонстрационную версию программы.

У вас есть несколько возможностей помочь потенциальному клиенту. Не упускайте из виду следующие рекомендации.

Убедитесь в том, что загрузку легко найти

Вероятно, вы полагаете, что найти страницу для загрузки демо-версии легко. Ведь вы знаете, где она находится, верно?

Не нужно предполагать. Найдите постороннего человека, попросите его посетить ваш веб-сайт и найти, где загружается демо-версия. Проследите за поиском и посмотрите, сколько времени он займет.

Сделайте демо-версию полнофункциональной

Лучшей демо-версией является сам продукт. Все продукты SourceGear состоят из единственного двоичного файла, загружаемого с сайта. Двоичный файл демо-версии ничем не отличается от двоичного файла конечного продукта. В нем работают все функции, но только в течение 30 дней. Чтобы купить программу, клиент просто вводит серийный номер; переустановка программы при этом не требуется.

Доведите до ума программу установки

Демо-версия — ваша возможность произвести первое хорошее впечатление. Крайне важно, чтобы она «просто работала». Если что-то пойдет не так, вероятно, клиент потеряет интерес к программе, и вам уже не удастся ничего исправить.

Сохраните анонимность клиента

Гиперссылка для загрузки демо-версии должна указывать непосредственно на двоичные файлы. Не заставляйте пользователя заполнять форму и оставлять контактные данные. Помните, что в пассивной модели главная роль отводится клиенту. Пусть он сам решит, когда сообщить вам о себе (если он вообще захочет это делать).

5. Отвечайте на вопросы клиентов

Я свято верю в важность качественной технической поддержки. Если у ваших клиентов возникают проблемы, задержитесь и помогите им. Более того, я считаю, что помощь клиентам является обязанностью каждого работника небольшой фирмы-разработчика.

В SourceGear все разработчики так или иначе помогают клиентам. У нас имеется группа технической поддержки «уровня 1», у которой поддержка является основной рабочей обязанностью. Но когда группа уровня 1 не справляется с работой или проблема оказывается слишком сложной, все разработчики готовы помочь с поддержкой «уровня 2». Нашим клиентам нравится, что при возникновении проблемы они могут поговорить с человеком, который написал проблемный код.

За несколькими редкими исключениями любой работник в штате должен быть готов приостановить свою текущую работу и помочь клиенту по мере надобности. Важным аспектом пассивной модели продаж является точно такое же отношение к потенциальным клиентам.

6. Предоставьте место для встреч

Потенциальные покупатели хотят иметь возможность поговорить с текущими клиентами о вас и о вашем продукте. На первый взгляд концепция выглядит устрашающе. А вдруг кому-то из клиентов ваш продукт чем-то не понравился? Надо ли, чтобы потенциальные покупатели общались с людьми, которые могут сказать что-то негативное?

Да, надо. Это пассивная модель продаж, и ведущая роль в ней принадлежит клиенту. Следует не только дать возможность потенциальным клиентам общаться с пользователями, но и предоставить им для этого место.

Жаль, что не все разработчики следуют этой рекомендации. В прошлом году я купил «Шевроле Аваланш» у фирмы, находящейся в моем районе. Подумайте, как было бы здорово, если бы продавец предоставил возможность поговорить с его прошлыми покупателями? Прежде чем я приму окончательное решение, он отводит меня в специальную комнату, где собрались все, кто когда-либо покупал «Аваланш» у этого конкретного продавца. Я немедленно начинаю задавать вопросы: понравилась ли вам машина? А вода не затекает? Не надо ли установить более мощный двигатель? Как насчет продавца, что это за тип? Не врал ли он вам?

Независимо от полученных ответов одно очевидно: этот продавец произвел на меня впечатление. Он не боится своих прошлых действий. Он верит в качество продукта и уровень поддержки, которую он предоставляет. Ему нечего скрывать.

Очевидно, продавец автомашин не сможет предоставить такую услугу, но для небольшой фирмы-разработчика это совсем несложно.

SourceGear поддерживает веб-форум, на котором пользователи и потенциальные покупатели могут общаться с нами и друг с другом. Посетители сайта свободно выражают свое мнение. Если клиент жалуется на нас (SourceGear) или на наши продукты, мы не стираем его комментарии.

Потенциальные клиенты часто посещают сайт и задают вопросы другим пользователям. Если кто-то из пользователей нашего продукта недоволен, вероятно, мы это заслужили. Вместо того, чтобы пытаться скрыть правду, мы попытаемся решить проблему.

Иногда такой подход связан с изрядными хлопотами, однако он обеспечивает механизм обратной связи, который заставляет нас постоянно совершенствовать наш продукт и помогать клиентам. Потенциальные покупатели это видят.

7. Упростите покупку через Web

Последним этапом в заполнении промежутка является тот момент, когда клиент отдает вам деньги, а вы отдаете ему программу. Как и на всех остальных этапах, основная роль здесь отводится клиенту, но вы обязаны упростить для него эту процедуру. Существует несколько различных способов создания электронных магазинов. Многие компании предлагают управлять магазином за вас. Также можно купить один из множества готовых программных пакетов. Я не обладаю достаточным опытом, чтобы порекомендовать какой-либо из этих вариантов, потому

что мы (SourceGear) всегда самостоятельно писали программное обеспечение своих электронных магазинов.

Одна из причин для самостоятельной разработки электронного магазина — возможность полной настройки процесса работы пользователя на сайте. Мы всегда пытаемся по возможности упростить приобретение своих продуктов. Наш электронный магазин должен немедленно генерировать серийные номера и отсылать их клиентам. Какой бы из вариантов вы ни выбрали, следующие рекомендации помогут вам упростить работу клиента на сайте.

Не заставляйте клиентов регистрироваться

Конечно, вашим клиентам совершенно не хочется запоминать еще одно имя с паролем.

Так ли уж необходимо, чтобы ваш электронный магазин создавал учетную запись для каждого, кто в нем что-то купил? Вероятно, нет.

Разве нельзя просто получить деньги и выдать покупателю программу? Наверное, можно.

Не создавайте корзину

По-моему, Amazon[1] слишком сильно влияет на наши представления об электронной коммерции. На мой взгляд, в Amazon совершенно замечательно реализована покупательская корзина. Она обладает исключительно мощными возможностями, и в то же время чрезвычайно проста. Так мы убеждаем себя, что наши электронные магазины должны быть такими же классными, как у Amazon, но на самом деле это не так. Amazon — настоящий электронный магазин, и метафора покупательской корзины в нем оправдана. Ассортимент Amazon огромен и включает неимоверное количество продуктов. Это очень приятное место. Вполне разумно, что покупатель захочет спокойно побродить по магазину, понемногу отбирая различные продукты, и остановиться на кассе, когда придет время оплатить покупки.

Вашей маленькой фирме такой масштаб попросту не нужен. Она больше напоминает фургончик для продажи гамбургеров, нежели универсам. Вы продаете ограниченное количество продуктов (а может быть, всего один). Клиенты не намерены озираться по сторонам и обозревать богатство вашего ассортимента. Они пришли купить гамбургер и не могут понять, зачем им вооружаться огромной корзиной и идти за полквартала, чтобы оплатить покупки.

Я говорю, руководствуясь собственным опытом и ошибками. До недавнего времени система электронных продаж SourceGear представляла собой крайне жалкое подобие Amazon. Произошло капитальное обновление, мы убрали корзину и упростили весь процесс заказа до единственной формы. Теперь все делается гораздо проще.

Немедленно предоставьте продукт покупателю

Не заставляйте своих клиентов ждать получения носителя или документации, чтобы они могли начать работу. Сразу же после того, как клиент разместит заказ,

[1] См. http://www.amazon.com.

дайте ему возможность загрузить двоичные файлы и немедленно начать работу с программой. А еще лучше — сообщите серийный номер для активации демо-версии, уже используемой им.

Но мы не можем так поступить!

А почему?

Я знаю, что многие не согласятся с моим мнением, изложенным в статье. Довериться клиенту страшно. Если вам не нравится то, что я написал, по крайней мере серьезно поразмыслите над следующим: разве мое описание не соответствует тому, как бы вы хотели себя чувствовать на месте клиента?

А если соответствует, то почему бы вам не поступить с клиентами именно так?

Почему бы и нет?

Никто не идеален

В каждой семинарии и в каждой религиозной школе будущих проповедников учат «проповедовать над собой». В конце концов, духовные пастыри — тоже люди. Как и у всех нас, у них есть свои проблемы. Чтобы каждое воскресенье вставать перед прихожанами и говорить о том, как жить лучше, нужно немалое мужество. Если бы совершенство было обязательным требованием для этой работы, то церковная кафедра всегда оставалась бы пустой.

Я сталкиваюсь с аналогичной проблемой в своих работах, но особенно в этой статье. Несколько раз я приводил нашу компанию в качестве примера, однако мы очень далеки от совершенства. Наши демо-версии не всегда работают. В электронном магазине иногда происходят странные вещи. Иногда мы слишком медленно реагируем на обращения в службу технической поддержки. Для большинства мелких фирм-разработчиков модель пассивных продаж является хорошим способом закрыть промежуток между покупателем и продуктом. Поручите ведущую роль клиенту, но придвиньте продукт как можно ближе к нему, чтобы по возможности упростить сокращение промежутка.

Эрик Синк

Опасности найма

Уволить работника всегда намного, намного труднее, чем не нанимать его. Существует множество людей, хорошо справляющихся со своей работой на целых 90 %. И хотя нам хотелось бы поручить эту работу тому, кто справится с ней на все 100 %, «работник на 90 %» не заслуживает увольнения, и компания продолжает работать с неполной эффективностью.

В малых фирмах правильный подбор персонала играет исключительно важную роль. В группе из пяти человек всего один плохой работник способен буквально парализовать всю группу.

К тому же «работник на 90 %» порождает другие проблемы. Допустим, он чуть-чуть запаздывает со сдачей своего компонента; последний откладывается До Выхода Следующей Версии, которая не выходит никогда, потому что ваша фирма к этому времени уже разоряется. Нельзя сказать, что работник никуда не годился — просто ему потребовалось 10 месяцев вместо 9. А может быть, секретарь, которому положено отвечать на телефонные звонки, недостаточно хорошо справляется с общением с потенциальными клиентами, и вы решаете нанять специалиста. Так вместо одного человека в штате появляются двое.

Или нанятый вами программист-новичок оказывается «так себе». Написанный им код выглядит на 90 % правильным, но содержит массу скрытых ошибок, так что его приходится отлаживать от начала и до конца, а старший программист вынужден переписывать и переделывать каждую строку кода, чтобы избавиться от хронической утечки памяти. Такой работник превращается в обузу; он поглощает чужое рабочее время, но при этом он недостаточно плох для увольнения. Кроме того, вы заставили его вместе с семьей сняться с места и лететь за 3000 миль, чтобы он мог работать на вас. Его дети только начинают привыкать к новой школе, супруга снова попала в больницу, они с трудом сводят концы с концами, и у вас рука не поднимется уволить его только за то, что он работает на 90 % вместо 100 %.

Гораздо проще не нанимать неподходящих людей, чем потом избавляться от них. — *Ред.*

Несколько месяцев назад я написал в MSDN статью под названием «Делайте побольше ошибок»[1]. Это была одна из самых популярных статей, когда-либо

[1] См. http://software.ericsink.com/bos/Make_More_Mistakes.html.

написанных мной. Похоже, людям понравилось читать о моих неудачах. Все мы просто люди, и нас восхищают чужие ошибки.

Во многих откликах на эту статью мне часто задавали один и тот же вопрос: почему я не упомянул ошибки с подбором персонала? «Эрик, возможно ли, чтобы ты никогда не совершал ошибок при найме?».

Au contraire, я их совершил предостаточно. Но эти истории я бы предпочел не рассказывать. Одно дело — публично признаться в своем идиотизме, и совсем другое — рассказать что-то такое, что может обидеть других. Тем не менее, принять решение о найме непросто; думаю, я узнал достаточно много, чтобы сказать что-то осмысленное по этой теме.

Я начну с четырех общих рекомендаций по поводу решений о найме. Статья завершается замечаниями по поводу специфических проблем, возникающих при найме разработчиков.

1. Нанимайте при возникновении необходимости, а не до нее

Принимая решение о найме, убедитесь в том, что это действительно необходимо. Мое правило для малых фирм-разработчиков гласит: не создавайте новую штатную единицу, пока необходимость такого шага не станет абсолютно очевидной.

В других ситуациях часто бывает разумно применять «опережающий» наем, рассчитанный на будущий рост. Многие венчурные компании работают подобным образом. Инвесторы дают вам миллионы долларов вовсе не потому, что они предпочитают хранить деньги в вашем банке вместо своего. Они ожидают стремительного роста вашей компании, который часто сопряжен с наймом дополнительного персонала. Но в небольшой компании, финансируемой из собственной прибыли, нанимать работника до того, когда необходимость этого шага станет абсолютно очевидной, почти всегда неправильно.

Совершить эту ошибку очень легко. Допустим, через восемь недель ожидается выход версии 7.0 вашего продукта. Ожидается множество новых заказов, и вы решаете нанять дополнительного человека в службу работы с клиентами, чтобы быть готовым к шквалу телефонных звонков.

Более правильное решение: пусть звонками займется кто-нибудь из существующего штата, или займитесь ими сами. Не увеличивайте расходы на зарплату, пока вы не будете на 100 % уверены в необходимости постоянного присутствия дополнительного работника.

Несколько лет назад я решил радикально подойти к задаче роста SourceGear «на следующий уровень». На работу было принято несколько новых людей, в том числе специалист по работе с персоналом. Мы убедили себя, что наша компания будет развиваться так быстро, что нам потребуется специальный человек. На эту должность был нанят первоклассный специалист — назовем ее Вилма.

Вилма была и до сих пор остается моим хорошим другом. Она хорошо поработала на SourceGear.

Но факт остается фактом: наша компания была недостаточно большой, чтобы ей требовался отдельный специалист по работе с персоналом. Мы знали об этом,

но рассчитывали сформировать штат с учетом ожидаемого роста. А затем мыльный пузырь Интернет-компаний лопнул, и компания SourceGear так и не выросла до предполагаемых размеров.

2. Помните, что наем — это лотерея

Наем — это лотерея. Оценивая кандидата, мы фактически пытаемся предсказать, сможет ли он успешно справиться со своей должностью. Мы пытаемся узнать будущее, однако у нас нет ни пророков, ни Оракула[1]. Следовательно, при оценке необходимо использовать различные признаки, которые, по нашему мнению, связаны с будущим успехом. Мы изучаем резюме, проверяем образование и рекомендации.

Но стопроцентной гарантии быть не может. Иногда все признаки положительны, но работник не справляется. В прошлом году я помогал благотворительной организации нанять нового работника. Мы нашли кандидата с исключительно надежным резюме (назовем его Уилбур).

С Уилбуром провели продолжительное собеседование. Мы проверили его рекомендации. Бесспорно, он обладал опытом, необходимым для работы. Решение казалось очевидным, поэтому он был принят. Вскоре после того, как Уилбур приступил к работе, началось что-то невообразимое. Неужели это тот человек, которого мы нанимали? Отношения между ним и группой были просто кошмарными. Несомненно, Уилбур был умным человеком с незаурядными способностями, но ситуация попросту не сложилась.

С другой стороны, иногда мы упускаем замечательного работника из-за признаков, указывающих в неверном направлении. Как правило, такие случаи остаются неизвестными. Кандидат получает отказ, и о дальнейшей его судьбе ничего не известно. Некоторые из них делают успешную карьеру.

3. Знайте законы

В Соединенных Штатах (и вероятно, в любой другой стране) существуют законы, которые необходимо изучить перед тем, как приступать к процессу найма. Это и федеральные законы, и законы штатов, и местные постановления. Я не юрист, поэтому я даже не буду пытаться объяснять эти законы, но очень важно, чтобы вы хорошо ориентировались в этих законах.

И еще одно замечание: даже если выяснится, что законы не распространяются на вашу компанию из-за ее малых размеров, все равно стоит выделить время на их изучение и по возможности следовать им. Как правило, соблюдение законов сыграет положительную роль в принятии решений.

4. Прислушивайтесь к разным мнениям

В этой области действует общий принцип: хорошие решения принимаются с учетом нескольких разных точек зрения. Если вы хотите последовательно принимать

---

[1] См. http://www.neoandtrinity.net/oracle.html.

самые худшие решения в области найма, принимайте их самостоятельно и не прислушивайтесь к чьему-либо мнению.

Но если вы хотите принимать мудрые решения, ознакомьтесь с разными мнениями и точками зрения. В своих решениях из области найма я обязательно учитываю мнение хотя бы одной женщины, работающей в моей компании.

Простой факт: в отрасли разработки программного обеспечения работает гораздо больше мужчин, чем женщин. Джулия Лерман[1] заметила, что на конференции Tech-Ed количество докладчиков по имени Брайан было больше, чем количество докладчиков-женщин. Наша область примерно на 90 % состоит из мужчин, и это означает, что мне приходится прикладывать дополнительные усилия для обеспечения баланса в области найма.

За прошедшие годы выявилась любопытная закономерность: многие неудачные решения из области найма, принятые мной, принимались без учета женского мнения.

В 1998 году компания SourceGear искала штатного специалиста в группу технической поддержки. Основное решение должен был принять я с одной из моих коллег по имени Мэри. Мы провели собеседование с несколькими кандидатами и разошлись во мнениях относительно того, кому из кандидатов следует отдать предпочтение. Я предложил Мэри принять решение самостоятельно; при этом я был уверен, что время докажет мою правоту. Но кандидат, выбранный Мэри, оказался одним из лучших работников за все время существования компании.

Стандартное правило найма программистов

Большинство статей, написанных на тему найма программистов, звучит более или менее одинаково. Как правило, такие статьи рекомендуют «нанимать только самых лучших». Признаюсь, я не в восторге от этого совета, потому что он звучит слишком неопределенно.

Пожалуйста, поймите меня правильно: я вовсе не советую намеренно выискивать посредственных программистов. Разумеется, все хотят нанимать только самых талантливых и опытных. В решениях о найме ставки высоки. Ваше решение повлияет на работу всей команды и ее отдельных участников. Как говорит Джоэл, «Отвергнуть хорошего кандидата гораздо лучше, чем принять плохого... Если у вас возникнут хотя бы малейшие сомнения — Не Нанимайте»[1].

Но стандартный совет все равно меня раздражает. Дело даже не столько в самом совете, сколько в том, что люди склонны понимать его неверно. Если применять его без дополнительных поправок, эта практика в основном формирует чувство собственного превосходства. Этот эффект особенно часто распространен среди программистов, поскольку элитарность так или иначе присуща нам. Когда мы слышим, что нанимать нужно только «самых лучших», этот совет проходит подсознательное преобразование:

---

[1] Tech-Ed — крупная конференция, регулярно проводимая компанией Microsoft. На этой конференции Microsoft информирует своих высокообразованных клиентов о своих последних разработках. — *Примеч. ред.*

«Самых лучших»? Но это же я! Я «самый лучший». Разумеется, я должен нанимать людей, таких же одаренных, таких умных и симпатичных, как я сам. Да и зачем засорять мою прекрасную команду всяким сбродом?

Само собой, такой подход создает не лучшие условия для принятия решений. Стандартное правило работает гораздо лучше, если понимать его несколько иначе:

«Я хочу сформировать как можно более эффективную команду. Нанимая дополнительного работника, я стремлюсь не только к численному расширению штата. Каждый нанимаемый человек должен улучшать мою команду в определенном отношении. Я вовсе не ищу такого же одаренного человека, как я сам. Скорее, мне нужен человек, одаренный более меня по крайней мере в одном важном направлении».

Худший начальник — тот, который ощущает угрозу со стороны своей команды. Сознательно или нет, он опасается «самых лучших», и поэтому постоянно нанимает людей, на фоне которых он будет смотреться выигрышно.

Наверное, в крупной компании с таким подходом можно прожить. Я сильно подозреваю, что Лохматый Начальник в комиксах про Дилберта был срисован с натуры.

Но в мире мелких фирм-разработчиков дело обстоит совершенно иначе. Если вы являетесь основателем или «главным гуру» в маленькой фирме, постарайтесь осторожно, честно и объективно взглянуть на самого себя. Если вы относитесь к числу людей, чувствующих угрозу со стороны собственных работников, остановитесь и задумайтесь. Пока вам не удастся решить эту проблему, шансы на построение эффективной команды будут равны нулю.

Истинный смысл стандартного правила заключается не в том, чтобы потешить наше самолюбие — оно должно напоминать, чтобы мы не боялись искать лучших работников. И все же необходимо более точно выяснить, что на самом деле означает слово «лучший».

Ищите людей, склонных к самоанализу

«Самые лучшие» работники никогда не перестают учиться.

Одним из важнейших критериев при оценке кандидатов я считаю то, что я про себя называю «первой производной». Учится ли этот человек? Движется ли он вперед или стоит на месте? (Некоторые мои размышления на эту тему опубликованы в статье «Career Calculus» в моем блоге[1]).

Люди, серьезно настроившиеся на свой будущий успех, с большой вероятностью добиваются успеха. Часто этот настрой является самым сильным признаком для принятия решений при найме.

Это не означает, что нанимать нужно только тех людей, которые желают добиться успеха. Все люди хотят добиться успеха. Я советую нанимать людей, серьезно относящихся к постоянному обучению. Такие люди не тратят время на попытки убедить вас в том, как много они знают. Они сосредоточены не на прошлом, а не будущем. Пока вы проводите с ними собеседование, они проводят собеседование с вами, пытаясь узнать, чему они смогут у вас научиться.

---

[1] См. http://www.joelonsoftware.com/articles/fog0000000073.html.

Как найти такого человека? По одному хорошо заметному признаку: люди, настроенные на постоянное обучение, хорошо знают, чего они не знают. Они знают свои слабости и не боятся говорить о них.

На собеседованиях кандидату часто предлагается описать свою основную слабость. Хотя этот вопрос до ужаса традиционен, он мне нравится.

К сожалению, многие кандидаты пытаются уклониться от ответа. Они идут в книжный магазин и покупают книгу о прохождении собеседования. Книга предупреждает, что я задам им этот вопрос, и предлагает «творческие» способы уклониться от искреннего ответа:

- Иногда я слишком много работаю.
- Иногда мое внимание к деталям раздражает других участников группы.

Когда я прошу кандидата рассказать о своих слабостях, я надеюсь на умный, искренний и уверенный ответ. Когда я слышу, что кандидат признает свою слабость, это производит впечатление. Но если кандидат дает уклончивый ответ, взятый прямо из книги, я начинаю думать о следующем кандидате.

Нанимайте разработчиков, а не программистов

В мелкой фирме «самыми лучшими» программистами являются те, которые не ограничиваются собственно программированием. Старайтесь нанимать разработчиков, а не программистов. Хотя эти слова часто используются как синонимы, я их различаю. Речь идет о различиях между простым программированием и участием в группе работы над продуктом. Приведу цитату из статьи, которую я написал на эту тему в своем блоге:

«В этой статье «программистом» называется тот, кто занимается исключительно кодированием новых функций и [если повезет] исправлением ошибок. Программисты не пишут спецификации. Они не создают автоматизированные контрольные примеры. Они не помогают поддерживать автоматизированные системы сборки в актуальном состоянии. Они не помогают клиентами решать технические проблемы. Они не помогают писать документацию, не участвуют в тестировании и даже не читают код. Все, что они делают, — это написание нового кода. В мелкой фирме таких людей держать не стоит.

Вместо «программистов» (людей, специализирующихся на написании кода) вам необходимы «разработчики» (люди, вносящие многосторонний вклад в успех продукта)[1]».

Что же означает стандартное правило? Какой именно атрибут нужно измерить, чтобы определить, является ли кандидат «самым лучшим»?

Обычно это правило понимается применительно только к навыкам кодирования. Но по-настоящему хорошие программисты отличаются сообразительностью. Они понимают то, чему обычно не учат, и могут работать раз в 10 эффективнее среднего

---

[1] См. http://software.ericsink.com/Career_Calculus.html.

программиста. Конечно, было бы разумно поискать одну из таких «десятикратных» личностей, особенно в больших организациях, где специалисты вроде «чистых» программистов оказываются вполне уместными. Но в маленькой фирме нужна универсальность. Нередко требуется, чтобы участники команд выполняли несколько функций, не ограничиваясь написанием кода. В таких случаях очень важно найти лучшего разработчика, и этот человек вовсе не обязательно окажется лучшим программистом.

О важности образования

Люди с хорошим общетехническим образованием часто оказываются «самыми лучшими» разработчиками.

Тема образования часто вызывает споры. Где-то в Интернете всегда найдется форум или чат, где люди спорят, какое образование действительно необходимо разработчику. Споры идут день и ночь напролет, 365 дней в году, но ответ на этот вопрос так и не был найден. И никогда не будет. Помните, что наем — это лотерея. Наличие образования является признаком, который может использоваться для прогнозирования успеха, но этот признак не всегда точен.

Два лучших разработчика SourceGear не имеют диплома. Один из них — превосходный программист, который постепенно становится превосходным разработчиком. Другой — превосходный разработчик, который постепенно становится превосходным программистом.

И все же я продолжаю сортировать анкеты по уровню образования. Эти два разработчика — исключения из правила. Весь мой личный опыт говорит о том, что диплом является полезным признаком будущего успеха. Нанимая разработчика, я хочу видеть степень бакалавра одного из признанных учебных заведений. Да, да, среди моих работников имеются два очевидных контр-примера. Но я еще раз скажу, что наем — это лотерея. И все же, на мой взгляд, степень бакалавра от Иллинойского университета или Стэнфорда повышает вероятность получить успешного работника.

Избыток образования — тревожный признак

С другой стороны, когда я вижу диплом доктора философии в области информационных технологий, это меня настораживает.

Университеты не учат, как стать разработчиком. Они даже не учат, как стать программистом. Университеты делают из своих студентов специалистов по информационным технологиям. Формирование программиста или даже разработчика обычно предоставляется студенту для самостоятельной работы.

Степень бакалавра закладывает прочный общетехнический фундамент и свидетельствует, что студент смог пройти курс обучения. Это важно, но когда речь заходит о конкретном наборе навыков, необходимых для небольшой фирмы, эффективность дальнейшего обучения начинает убывать с первого дня аспирантуры.

Многие люди страшно обижаются на это мнение. Пожалуйста, поймите меня правильно: я очень уважаю людей с дипломом доктора философии. Для получения докторской степени необходимы огромная дисциплина, ум и стремление. Я восхищаюсь этими качествами и серьезно сомневаюсь в том, что мне самому удалось бы получить докторскую степень.

Но я все равно считаю, что эти качества не являются навыками, необходимыми для небольшой фирмы-разработчика. Для создания коммерческого продукта необходимы несколько иные виды дисциплины, ума и стремления. Эти качества аналогичны «докторским качествам», и все же сильно отличаются от них.

Более того, я считаю, что люди, одновременно обладающие качествами обоих видов, встречаются крайне редко. Когда я вижу человека с докторским дипломом, я твердо уверен в том, что он обладает «докторскими качествами»... и поэтому мне кажется маловероятным, чтобы он также обладал качествами разработчика коммерческих продуктов.

Конечно, я могу я ошибаться, и все же в очередной раз скажу, что наем — это лотерея. Мы используем признаки для прогнозирования будущего успеха кандидата, но эти признаки не всегда верны. У каждого правила существуют свои исключения, и, следуя теории вероятности, я упускаю эти исключения. Это весьма прискорбно, потому что доктор философии с талантами разработчика был бы невероятно ценным работником. Допустим, к примеру, что я получил резюме от человека с докторской степенью в области информационных технологий и несколькими годами работы в команде Adobe Photoshop. Естественно, я приглашу этого человека на собеседование. Наверное, не найдется коммерческого продукта, которым бы я восхищался более, чем Photoshop. С докторской степенью или без нее, этот человек очевидно обладает качествами разработчика коммерческих продуктов. Степень не является изначально негативным фактором. Это всего лишь один из признаков, и иногда он дает сбой.

Просмотрите код

Хотя я уделяю особое внимание аспектам разработки программного обеспечения, не связанным с программированием, код тоже очень важен. «Самые лучшие» разработчики просто обязаны быть очень хорошими программистами.

Не бойтесь просматривать код. Проводя собеседование с разработчиком, попросите показать образцы кода. Попросите написать небольшой фрагмент программы прямо во время собеседования. Один из моих любимых вопросов, который я задаю многим кандидатам, — сколько строк кода они написали за всю свою карьеру? Ответы оказываются самыми разными. Одни кандидаты попросту не знают. Другие говорят, что вопрос дурацкий, и выдают обширное исследование, доказывающее, что «количество строк кода» не является хорошим показателем производительности работы программиста. Превосходно, они имеют право на собственное мнение, но вопрос мне все равно нравится. Я считаю, что квалификация программиста повышается по мере написания кода. Мне хочется знать, сколько кода написал кандидат.

Во время учебы в колледже я написал компилятор C — просто для развлечения. Он был написан на C «с нуля», с рекурсивной системой разбора и даже некоторыми

локальными оптимизациями. Компилятор был не очень быстрым, но компилировал свой исходный текст без ошибок. Я опубликовал свой компилятор на условиях лицензии GPL, но остался единственным человеком, который его когда-либо использовал.

Поступая на работу в Spyglass, я показал свой компилятор менеджеру по найму персонала. Я получил работу, и мой проект стал одним из факторов при принятии решения. По словам менеджера, при просмотре моего компилятора он увидел, что я уже порядком намучился с плохим кодом в своей системе, так что можно было предположить, что следующие сто тысяч строк будут относительно хорошими ☺

Спустя двенадцать лет я думаю, что практика найма людей, внесших заметный вклад в проекты сообщества открытых исходных кодов, вполне разумна. В этом случае мне даже не приходится просить примеры кода; я могу просто взять архив и прочитать его самостоятельно.

Однако доступность кода для просмотра отнюдь не является главной причиной, по которой мне хотелось бы видеть в резюме опыт участия в проектах с открытыми исходными кодами. Работа над такими проектами также кое-что говорит о человеке.

Стоит признать, что многие программисты руководствуются исключительно ненавистью к Microsoft. Независимо от вашего отношения к Microsoft, подобная мотивация вряд ли станет хорошей базой для успеха в любой работе, связанной с разработкой.

Но многие люди работают над проектами с открытыми исходными кодами просто потому, что им нравится программировать. Это их увлечение, причем далеко не худшее. Некоторые смотрят на такие проекты, как AbiWord[1] или ReactOS[2], и видят за ними лишь других людей, попусту тратящих свое время на клонирование коммерческих продуктов Microsoft. Соглашусь с тем, что такие проекты действительно не имели бы особого смысла, если бы им пытались найти коммерческое применение. Но типичный участник таких проектов программирует просто ради удовольствия. Телевизор — напрасная трата времени, а программирование — нет.

Люди, которым просто нравится писать программный код, часто оказываются «самыми лучшими» разработчиками.

Самые лучшие

Итак, стандартное правило работает вполне нормально, но от общих рекомендаций необходимо перейти к более конкретным. Чтобы подвести итог тому, о чем говорилось в предыдущих разделах, я предлагаю 10 вопросов, которые следует задать себе при рассмотрении кандидата на должность разработчика:

1. Способен ли этот кандидат сделать для группы то, что не сможет никто другой?
2. Находится ли он в процессе постоянного обучения?

---

[1] См. Sec http://www.abisource.com.

[2] См. http://www.reactos.com.

3. Знает ли этот кандидат о своих слабостях и может ли спокойно обсуждать их?
4. Насколько этот кандидат универсален и способен сделать «все, что потребуется», чтобы обеспечить коммерческий успех продукта?
5. Принадлежит ли кандидат к числу «десятикратных» программистов?
6. Обладает ли он степенью бакалавра, полученной в одном из уважаемых университетов?
7. Если кандидат обладает докторской степенью, имеются ли другие признаки, свидетельствующие о его способностях к разработке коммерческих продуктов?
8. Имеется ли у кандидата опыт работы в группах, занимавшихся разработкой коммерческих продуктов?
9. Может ли кандидат представить примеры хорошего кода?
10. Любит ли кандидат программирование настолько, чтобы писать код в свободное время?

Положительный ответ на все 10 вопросов не обязателен. Я даже не собираюсь указывать максимальное количество положительных ответов, необходимых для принятия кандидата. Наем — это лотерея, и каждый вопрос может послужить признаком для оценки пригодности кандидата.

В конечном счете, любое решение из области найма осуществляется волевым решением, и никакие гарантии здесь невозможны. И все же если уделить внимание этим вопросам, они повысят вероятность того, что вам не придется позднее пожалеть о принятом решении.

Аарон Шварц

Вариации на тему PowerPoint

В своих докладах я стараюсь избегать унылых презентаций PowerPoint с рядами маркеров-«пулек». Вместо них я использую фотографии классных актеров (вернее симпатичных — Дженифер Энистон, Брэд Питт и т. д.), собачек, рассказываю анекдоты и т. д.; на мой взгляд, все перечисление должно ограничиваться одним слайдом, содержащим не более трех маркеров. Конечно, когда докладчик так занят анекдотами и демонстрацией портретов актеров, на перечисления времени обычно не остается.

В 2004 г. Эдвард Р. Тафт (Edward R. Tufte), известный своими блестящими книгами по визуальному отображению информации, решил, что ему хватило PowerPoint на всю оставшуюся жизнь. Он начал кампанию за избавление мира от этого зла. «К сожалению, — писал Тафт, — слайдовые программы часто снижают аналитическое качество презентаций. В частности, популярные шаблоны PowerPoint (заготовки слайдов) обычно ослабляют вербальное и пространственное воздействие, и почти всегда портят статистический анализ». В его превосходном очерке «Стиль представления материала в PowerPoint» проблема четко излагается всего на 28 страницах.

Аарон подготовил сводку. — *Ред.*

«Стиль представления материала в PowerPoint» Эдварда Р. Тафта[1] в форме презентации PowerPoint

Обзор

- PowerPoint стал стандартом...
- ...но плохим.
- Почему?

Стиль представления материала

- Ориентирован на докладчика
- Страдает аудитория и содержание
- Низкое разрешение

---

[1] Tufte, Edward. The Cognitive Style of PowerPoint. Cheshire, CT: Graphics Press, 2003.

- Чрезмерно иерархическая структура
- Излишнее внимание форме

Низкое разрешение

- Почти всегда в ущерб содержанию
- Лишь немногим лучше пропаганды в «Правде» образца 1982 г.

Ослабляет мыслительный процесс

- Маркированные списки рассчитаны на дураков
  - Слишком общие
  - Не отражают логические связи
  - Не отражают предположения

[Вставка: презентация с анализом катастрофы «Колумбии»[1]]

Глубокая иерархия

- Часто доходит до шести уровней
- А Фейнману было достаточно двух

Почему?

- Базируется на представлениях корпорации-разработчика
  - Большая бюрократия
  - Занимается программированием компьютеров
    - Глубокая иерархия
  - Маркетинг
    - Неверный выбор направления
    - Пристрастие к лозунгам
    - Преувеличение

Почему? (продолжение)

- Что могло бы быть хуже?
  - Сталин?
- Бесцеремонность
  - Материал приходится укладывать в навязанную модель
- За основу взята модель «памятник вождю на пьедестале»

Что делать?

- Лучший выход: хорошее обучение
  - Объяснение, разумные доводы и т. д.
  - Источники, заслуживающие доверия

PowerPoint в школах

- Нехорошо!

---

[1] См. http://www.edwardtufte.com/bboard/q-and-a-fetch-msg?msg_id=0000Rs&topic_id=1&topic=Ask%20E%2cT%2e.

- Нужно найти замену
  - Вариант: научить детей курить
  - Лучше: закрыть школу
  - Еще лучше: написать очерк с картинками

[Вставка: презентация PowerPoint со сражением при Геттисберге[1]]

[Вставка: а если бы в PowerPoint отображался уровень смертности от рака?]

Таблицы стилей

- Корпоративный логотип
  - Указывается название отдела
  - А не имена реальных людей (стесняются, что ли? — *А. Ш.*)
- Имитация букваря для 6-летних
- Слабое оформление
- Логическое деление затрудняет сравнения
- Бесполезные таблицы

Мировое господство

- Печатные презентации: 50 слайдов = 1 страница справочника врача
- Электронные презентации: 20 % объема популярных веб-сайтов
- Наихудшее соотношение «сигнал/шум»

О последовательности

- Маркеры гипнотизируют аудиторию
- Резкие переходы, как при некачественном видеомонтаже
- Раздача печатных материалов позволит аудитории контролировать порядок и темп

Что делать?

- Немедленный отзыв продукта по всему миру
- Аналог экспериментального лекарства с побочными эффектами

Улучшение презентаций

- Улучшение содержимого
- Раздача печатных материалов
- Не создавать бессмысленные презентации

Завершающие мысли — *А. Ш.*

- Хороший очерк
- Купить много экземпляров
- Раздать докладчикам, использующим PowerPoint

---

[1] См. http://www.norvig.com/Gettysburg.

why the lucky stiff[1]

Краткая (и будем надеяться, приятная) экскурсия по языку Ruby

В славную эпоху компьютеров (то есть когда я был подростком) существовала традиция писать забавные, легкие, приятные книги по программированию, из которых можно было действительно узнать что-то полезное.

К Apple прилагалось руководство пользователя, полное юмора в стиле Monty Python; в частности, при описании обращения с дискетами автор сурово предупреждал, что дискеты никогда не следует подшивать степлером к бумагам. Я три недели хихикал над этой шуткой и читал ее вслух каждому, кто соглашался послушать. Не помню, почему. Вероятно, моему юному уму это казалось забавным.

«The Little LISPer» Дэниела Фридмана (Daniel Friedman) был изящным, кристально чистым шедевром, возможно, лучшим из когда-либо написанных руководств по программированию — и это при том, что книга учила языку, который, честно говоря, оказывается не под силу большинству читателей.

И язык C никогда бы не стал стандартом, если бы не знаменитая книга Брайана Кернигана и Денниса Ричи («The C Programming Language», Prentice-Hall, 1978), которая была примером четкой подачи материала и блестящей методики обучения, хотя в ней не было ни одной мультяшной лисы или другого животного.

К сожалению, ситуация с изданием компьютерной литературы изменилась к худшему, когда издатели заметили, что бестселлером обычно оказывается книга с самым толстым корешком. Ошарашенные покупатели, которым приходилось выбирать между изучением Java и сменой работы, шли в соседний магазин и видели перед собой целую Стену из Книг по Java. Был только один разумный способ выбора — взять самую толстую книгу. Так мы пришли к тому, что имеем: ужасные книги с красными обложками, которые ничему вас не учат.

К счастью, why the lucky stiff с парой рисованных лис в этом кратком описании языка Ruby воскрешает великую традицию развлекательных учебников по программированию. — *Ред.*

---

why the lucky stiff (также известен под именами why или _why) — псевдоним писателя, музыканта, художника и программиста, наиболее известного как автор языка программирования Ruby. Фамилия этого писателя — Мальский (Malsky), имя же его до сих пор остается неизвестным (см. http://en.wikipedia.org/wiki/Why_the_lucky_stiff). — *Примеч. ред.*

Да, мне будут помогать эти двое. Кажется, от шерсти у меня разыгралась астма, и мне нужно глотнуть свежего воздуха. Я сейчас вернусь.

Мне сказали, что к этой главе следует прилагать платок или что-нибудь другое, чем можно вытереть лицо, когда по нему будут стекать струи пота.

В самом деле, нам предстоит стремительно промчаться по всему языку. Это будет выглядеть так, словно мы чиркаем спичку за спичкой так быстро, как это только возможно.

Язык... Да, просто язык

Моя совесть не позволяет мне назвать Ruby *компьютерным* языком. Это подразумевало бы, что Ruby работает главным образом на условиях компьютера. Что он проектировался прежде всего в интересах компьютеров. Что мы, программисты, здесь чужие, желающие прописаться в компьютерном царстве. Что на этом языке говорит компьютер, а мы всего лишь переводчики.

Но как назвать язык, на котором мыслит ваш мозг? Когда вы начинаете использовать слова и конструкции языка для выражения своих мыслей? Компьютеры этого не умеют. Разве этот язык может быть компьютерным? Это наш язык, мы говорим на нем!

Называть Ruby компьютерным языком было бы несправедливо. Это язык наших мыслей.

Прочитайте вслух следующую строку:

```
5.times { print "Odelay!" }
```

В английском языке знаки препинания (точки, восклицательные знаки, скобки) не читаются. Знаки препинания добавляют смысл и дают визуальные подсказки по поводу того, что хотел сказать автор данным предложением. Итак, давайте прочитаем эту строку в следующем виде:

```
Пять раз вывести "Odelay!"
```

Именно это и делает наша маленькая программа на Ruby. На экране компьютера будет пять раз выведено испанское восклицание, искаженное Беком[1].

Прочитайте вслух следующую строку:

```
exit unless "restaurant" include? "aura"
```

Здесь выполняется простейшая проверка на реальных данных. Наша программа завершит работу (exit), если слово restaurant не содержит (include) слово aura.

Вы когда-нибудь видели язык программирования, в котором бы так эффективно использовались вопросительные знаки? В Ruby некоторые знаки препинания (восклицательные и вопросительные) используются для повышения удобочитаемости кода. В этом фрагменте мы фактически задаем вопрос, так почему бы не сделать этот факт очевидным?

---

[1] См. http://en.wikipedia.org/wiki/Odelay.

Прочитайте вслух следующую строку:

```
[toast. cheese. wine].each { |food| eat food }
```

Хотя этот фрагмент не столь легко читается и менее напоминает привычные предложения, чем предыдущие примеры, я все равно рекомендую прочитать его вслух. Хотя Ruby иногда читается как английский текст, в отдельных случаях он представляет собой *укороченный* текст. После полного перевода на английский эта конструкция читается так: для тостов (toast), сыра (cheese) и вина (wine), взять каждый вид пищи и съесть его.

Если запустить эту программу, она работать не будет. Ruby не знает, что такое «есть» (eat). К счастью, в Ruby можно добавлять новые слова. Ваши собственные действия. Ваши собственные объекты.

Вероятно, вы уже интересуетесь, как эти слова работают в сочетании друг с другом. Кому-то хочется знать, что означают эти точки и скобки. Вскоре мы обсудим разные части речи в Ruby.

А пока все, что вам необходимо знать, — что код Ruby строится из предложений. Это не совсем английские фразы, а скорее короткие наборы слов и знаков препинания, выражающие единую мысль. Из предложений складываются книги. Из них могут складываться страницы и даже целые повести — понятные не только человеку, но и компьютеру.

Части речи

Многие части речи в Ruby снабжаются визуальными признаками, по которым их проще узнать (на манер белой полосы на спине скунса или развевающегося белого шлейфа невесты). Знаки препинания и регистр символов помогут вашему мозгу разглядеть компоненты программы и испытать мгновенное чувство узнавания. Ваш разум будет часто вопить: *Эй, я знаю, что это такое!* Кроме того, вам будет чем похвастаться в разговорах с другими любителями Ruby.

Сейчас нас интересует внешний вид каждой из этих частей речи, а подробности будут изложены дальше. Я приведу краткие описания, но вы не обязаны полностью понимать мои пояснения. К концу главы вы сможете узнать любую составляющую программы на языке Ruby.

Переменные

Любое слово, записанное обычными строчными буквами, в Ruby является переменной. Переменные могут содержать буквы, цифры и символы подчеркивания.

Примеры переменных — `x`, `y`, `banana2` и `phone_a_quail`.

Переменная — это как прозвище. Помните, как в детстве вас дразнили «Вонючка Пит»? Люди говорили: «Эй, Вонючка Пит, иди сюда!» И все окружающие мгновенно понимали, что речь идет именно о вас.

Переменная тоже назначает «прозвище» для часто используемых данных.

Допустим, вы руководите сиротским приютом. И каждый раз, когда папаша Уорбек приходит, чтобы купить новых сироток, вы настаиваете на том, чтобы он заплатил вам сто двадцать один доллар и восемь центов за плюшевого мишку,

который скрашивал несчастному ребенку пребывание в вашем кошмарном заведении.

```
teddy_bear_fee = 121.08
```

Позднее, когда вы пробиваете чек (на навороченном кассовом аппарате, работающем на языке Ruby!), все затраты нужно сложить для получения общей суммы (total):

```
total = orphan_fee + teddy_bear_fee + gratuity
```

Конечно, запомнить прозвище-переменную проще, чем конкретное число.

Числа

Простейшую разновидность чисел составляют целые числа — последовательности цифр, которые могут начинаться со знака «плюс» или «минус».

Примеры чисел — 1, 23 и –10 000.

Запятые в числах недопустимы, но символы подчеркивания разрешены. Если вы захотите разделить разряды на группы, чтобы число было проще читать, воспользуйтесь символом подчеркивания:

```
population = 12_000_000_000
```

Дробные числа в Ruby называются вещественными. К этой категории относятся числа с десятичной точкой или записанные в научной (экспоненциальной) записи.

Примеры вещественных чисел — 3.14, -808.08 и 12.043e-04.

Строки

Строка представляет собой любую последовательность символов (букв, цифр, знаков препинания), заключенную в кавычки или апострофы.

Примеры строк — «sealab», '2021' и «These cartoons are hilarious!".

Символы, заключенные в кавычки или апострофы, хранятся вместе как единое целое. Представьте себе репортера, записывающего бессвязные откровения какой-нибудь знаменитости (скажем, Эврил Лавин).

```
avril_quote = "I'm a lot wiser. Now I know
what the business is like -- what you have
to do and how to work it."
```

Итак, раньше мы сохраняли число в переменной teddy_bear_fee, а сейчас сохраняем последовательность символов (строку) в переменной avril_quote.

Репортер передает свои записи издателям, у которых все оборудование работает на языке Ruby:

```
Tabloid.print oprah_quote
Tabloid.print avril_quote
Tabloid.print justin_timberlake_pix
```

Символические имена

Символические имена (symbols) — слова, внешне очень напоминающие переменные. Как и переменные, они могут содержать буквы, цифры или символы подчеркивания. Тем не менее, символические имена начинаются с двоеточия.

Примеры символических имен — :a, :b и :ponce_de_leon.

Символическое имя можно рассматривать как облегченную строку. Как правило, они используются при работе со строками, которые вы не собираетесь выводить на экран. Можно сказать, что символическое имя компьютеру легче переварить. Это как средство от лишней кислотности, а двоеточие — пузырьки, поднимающиеся из брюха вашего компьютера при переваривании символического имени. Ах... какое облегчение.

Константы

Константы, как и переменные, представляют собой слова, но начинаются с прописной буквы. Если переменные были существительными Ruby, то константы можно рассматривать как имена собственные.

Примеры констант: Time, Array и Shake_It_Like_A_Polaroid_Picture.

В английском языке имена собственные пишутся с большой буквы — Эмпайр Стейт Билдинг. Вы не можете просто передвинуть Эмпайр Стейт Билдинг на другое место, или решить, что с этого дня этим словом называется что-то другое. Таковы все имена собственные. Они обозначают что-то очень конкретное, что не изменяется с течением времени.

Так же дело обстоит и с константами: после того, как константе будет присвоено значение, она уже не изменяется.

```
EmpireStateBuilding = "350 5th Avenue, NYC, NY"
```

Если мы попытаемся изменить константу, Ruby пожалуется на наше нехорошее поведение. Такие вещи не поощряются.

Методы

Если переменные и константы представляют существительные Ruby, то методы соответствуют глаголам. Как правило, методы указываются после переменных и констант, и отделяются от них точкой. Мы уже видели примеры методов.

```
front_door.open
```

В этой строке open — имя метода. Это действие, операция, глагол. В некоторых ситуациях действия объединяются в цепочки:

```
front_door.open.close
```

Мы приказываем компьютеру открыть (open) парадную дверь, а затем немедленно закрыть ее (close).

```
front_door.is_open?
```

Это тоже операция: мы хотим, чтобы компьютер проверил состояние двери и выяснил, открыта ли она. Метод можно было бы назвать Door.test_to_see_if_its_open, но имя is_open? короче и ничем не хуже. Имена методов могут содержать как вопросительные, так и восклицательные знаки.

Аргументы методов

Для выполнения своей задачи методу также может понадобиться дополнительная информация. Например, если мы хотим, чтобы компьютер покрасил дверь, следует указать цвет.

Аргументы метода перечисляются после его имени. Обычно они заключаются в круглые скобки и разделяются запятыми.

```
front_door.paint( 3, :red )
```

Эта команда покрасит дверь тремя слоями красной краски (red).

Вызовы методов с аргументами также могут объединяться в цепочки:

```
front_door.paint( 3, :red ).dry( 30 ).close()
```

Эта команда красит дверь тремя слоями красной краски, выжидает 30 минут, а затем закрывает дверь. Хотя последний метод вызывается без аргументов, при желании все равно можно использовать круглые скобки. Впрочем, в данном случае скобки не нужны, поэтому на практике они обычно опускаются.

Некоторые методы (такие, как print) являются методами ядра. Эти методы постоянно используются в Ruby, и из-за их распространенности точка обычно не указывается:

```
print "See, no dot."
```

Методы классов

Как и методы, о которых говорилось выше (также называемые методами экземпляров), методы классов обычно следуют за именами переменных и констант. Вместо точки в этом случае используется удвоенное двоеточие:

```
Door::new( :oak )
```

Как нетрудно догадаться, метод класса new чаще всего используется для создания чего-либо. В этом примере мы просим Ruby создать для нас новую дверь (Door) из дуба (oak). Конечно, язык Ruby должен знать, как сделать дверь, и у него должны быть все необходимые материалы.

Глобальные переменные

Переменные, имена которых начинаются со знака доллара ($), являются глобальными.

Примеры глобальных переменных — $x, $1, $chunky и $CHunKY_bACOn.

Большинство переменных имеет временную природу. Некоторые части программы напоминают маленькие домики со своими собственными наборами переменных. В одном домике папой является коммивояжер Арчи, в другом — укротитель львов Питер, и т. д. В каждом доме понятие «папа» означает что-то свое.

При работе с глобальными переменными можно быть уверенным в том, что смысл переменной остается одним и тем же во всех домиках.

Глобальные переменные могут использоваться где угодно. Они никогда не выходят из поля зрения программы.

Переменные экземпляров

Переменные, имена которых начинаются со знака @, относятся к переменным экземпляров.

Примеры переменных экземпляров — @x, @y и @only_the_chunkiest_cut_of_bacon_I_have_ever_seen.

Переменные экземпляров часто используются для определения атрибутов чего-либо. Например, вы можете сообщить Ruby ширину двери (front_door), устанавливая значение переменной @width внутри объекта front_door.

Считайте, что знак @ означает «атрибут».

Переменные классов

Переменные, имена которых начинаются с двойного знака @, называются переменными классов.

Примеры переменных классов — @@x, @@y и @@i_will_take_your_chunky_bacon_and_raise_you_two.

Переменные классов тоже используются для определения атрибутов. Но вместо определения атрибута одного объекта, переменные классов задают атрибуты множества взаимосвязанных объектов. Если переменная экземпляра задает атрибут одной двери, то переменная класса задает атрибут всех существующих дверей.

Считайте, что префикс @@ означает «атрибут для всех». При изменении переменной класса изменяется атрибут не только одного объекта, а всех объектов сразу.

Блоки

Любой код, заключенный в фигурные скобки, называется блоком.

Пример блока:

```
{ print "Yes. I've used chunky bacon in my examples, but never again!" }
```

Блок группирует набор инструкций, чтобы в дальнейшем они использовались как единое целое. Фигурные скобки напоминают клешни краба, которые схватили сгруппированные команды и держат их вместе. Когда вы видите две клешни. помните, что содержащийся в них код является одним целым.

Фигурные скобки также могут заменяться словами do и end. Это удобно, если длина блока превышает одну строку:

```
do
print "Much better."
print "Ah. More space!"
print "My back was killin' me in those crab pincers."
end
```

Аргументы блоков

Аргументы блоков представляют собой набор переменных, заключенных между вертикальными чертами и разделенных запятыми:

Примеры аргументов блоков: |x|, |x,y| и |up, down, all_around|.

Аргументы блоков используются в начале блока:

```
{ |x,y| x + y }
```

В этом примере |x,y| — аргументы. За аргументами следует небольшой фрагмент кода. Выражение x + y суммирует два аргумента.

Мне нравится думать, что вертикальные черты изображают желоб, по которому скатываются переменные (x распласталось во все стороны, а y аккуратно скрестило

ноги). Желоб связывает блок с окружающим миром. Переменные передаются по этому желобу в блок.

Интервалы

Интервал состоит из двух значений, заключенных в круглые скобки и разделенных двумя или тремя точками.

Интервал (1..3) представляет числа от 1 до 3.

Интервал ('a'..'z') представляет строчные буквы алфавита.

Представьте себе интервал как аккордеон, сжатый для переноски. Круглые скобки изображают рукоятки на сторонах маленького, сложенного аккордеона, а точки — цепочку, которая удерживает аккордеон в сложенном состоянии.

Обычно в интервалах используются только две точки. Если присутствует третья точка, значит, последнее значение исключается из интервала.

Интервал (0...5) представляет числа от 0 до 4.

Когда вы видите третью точку, представьте, что аккордеон слегка приоткрылся — ровно настолько, чтобы выпустить одну ноту. Эта нота — последнее значение в интервале. Пусть она улетает в небо.

Массивы

Массив представляет собой список, элементы которого заключены в квадратные скобки и разделены запятыми:

[1, 2, 3] — массив чисел.

['coat', 'mittens', 'snowboard'] — массив строк.

Представьте себе гусеницу, втиснутую в ваш код. Две квадратные скобки не позволяют гусенице двигаться, чтобы вы могли следить за тем, где находится голова, а где — хвост. Запятые представляют ноги гусеницы между сочленениями ее тела.

Массив представляет набор значений, элементы которого хранятся в определенном порядке.

Хеши

Хеш (ассоциативный массив) представляет собой словарь, элементы которого перечисляются в фигурных скобках. Как известно, в словарях устанавливается связь между словами и их значениями. В Ruby для обозначения этой связи используются стрелки, состоящие из знака равенства и знака «больше».

Пример: {'a' => 'aardvark', 'b' => 'badger'}.

На этот раз фигурные скобки представляют изображения книг. Видите, как они похожи на открытые книги с загнувшимися краями страниц? Эти изгибы напоминают о том, как мы открываем и закрываем словарь.

Представьте, что на каждой странице словаря находится одно определение. Запятая изображает уголок страницы, которую необходимо перевернуть, чтобы увидеть следующее определение. На каждой странице располагается слово, за которым следует ссылка, указывающая на определение.

```
{
'name' => 'Peter',
```

```
'profession' => 'lion tamer',
'great love' => 'flannel'
}
```

Я сравниваю хеши со словарями вовсе не потому, что в хешах можно хранить только определения. В представленном примере в хеше хранится личная информация о Питере, укротителе львов. Основное сходство хеша со словарем — это прежде всего простота поиска.

В отличие от массивов, порядок следования элементов хеша не определен.

Регулярные выражения

Регулярное выражение представляет собой набор символов, заключенных между наклонными чертами.

Примеры: /ruby/, /[0-9]+/ и /^\d{3}-\d{3}-\d{4}/.

Регулярные выражения используются для поиска слов и отдельных участков текста. Наклонные черты по обе стороны выражения — это ограничители.

Представьте, что у вас имеется слово, к которому с обоих концов прикреплены булавки, и вы держите его в руках над книгой. Вы проводите словом над текстом, и когда оно оказывается рядом с совпадающим участком, слово начинает мигать. Вы прикалываете регулярное выражение к тексту, втыкая булавки по обе стороны от совпадения, и выражение начинает светиться буквами совпадающего слова.

Поиск по регулярному выражению происходит гораздо быстрее, чем вы водите руками над страницами. Ruby позволяет использовать регулярные выражения для проведения очень быстрого поиска в огромных объемах текста.

Операторы

Далее перечислены операторы, используемые в Ruby для вычислений и для сравнения величин. Просмотрите список; наверняка вы найдете в нем что-нибудь знакомое. Сами знаете, сложение — «+», вычитание — «–», и так далее.

```
** ! ~ * / % + - &
<< >> | ^ > >= < <= <=>
|| != =~ !~ && += -= =
.. ... not and or
```

Ключевые слова

> Ruby has a number of built-in words, imbued with meaning. These words cannot be used as variables or changed to suit your purposes. Some of these we've already discussed. They are in the safe house, my friend. You touch these and you'll be served an official syntax error.

```
alias   and     BEGIN   begin   break   case    class   def     defined
do      else    elsif   END     end     ensure  false   for     if
in      module  next    nil     not     or      redo    rescue  retry
return  self    super   then    true    undef   unless  until   when
while   yield
```

Вот и хорошо. Мы познакомились с основными компонентами языка Ruby. В следующих трех главах мы попробуем соединить их в рабочие фрагменты кода

Я рекомендую еще раз пробежаться по всем частям речи. Постарайтесь представить себе «общую картину». В следующем разделе я проверю ваше усердие.

Снова в школу

Я горжусь вами. Любой скажет, как высоко я вас ценю. Как я отношусь к этой великой анонимной личности, которая прокручивает и читает, прокручивает и читает... «Эти парни, — говорю я, — да, эти парни знают, чего они хотят. Я никогда...» И мне даже не удается закончить предложение из-за переполняющих меня чувств.

И мое сердце начинает ярко светиться под моей тонкой, прозрачной кожей, и только 10 кубиков JavaScript внутривенно приводят меня в чувство (я хорошо реагирую на токсины в крови).

Так что вы пока показали себя вполне достойно. Но сейчас я начинаю изображать бессердечного школьного учителя. Мне нужно видеть ваши хорошие отметки. Пока вы еще ничего серьезного не сделали, только много водили глазами по строчкам. Ну да, раньше вы еще немного почитали вслух. Теперь я хочу видеть, как вы усвоили этот материал.

Назовите вслух все части речи в этой строке:

```
5.times { print "Odelay!" }
```

Может быть, даже стоит закрыть этот абзац рукой, потому что ваши глаза сами попробуют подсмотреть ответ. Мы видим число 5, за которым следует метод .times. Далее идет первая «клешня», открывающая блок. Метод ядра print вызывается без точки, а за ним следует строка «Odelay!". Вторая «клешня» завершает блок.

Назовите вслух все части речи в этой строке:

```
exit unless "restaurant".include? "aura"
```

Как и print, метод exit является методом ядра. Если вы внимательно просмотрели длинный список ключевых слов, то заметили, что unless — это тоже ключевое слово. К строке «restaurant» жмется метод include?. И наконец, за ним следует строка «aura».

Назовите вслух все части речи в этой строке:

```
[toast, cheese, wine].each { |food| eat food }
```

Этот пример начинается с массива-«гусеницы». Массив содержит три переменные: toast, cheese и wine. За массивом следует вызов метода each.

Внутри блока аргумент блока food спускается вниз по желобу в блок. Затем метод eat использует аргумент блока, превратившийся в переменную food.

Еще раз просмотрите все примеры. Убедитесь в том, что вы узнали все использованные части речи. Каждая из них внешне отличается от других, не правда ли? Сделайте глубокий вдох и прижмите пальцы к вискам. Сейчас мы разберем нашу первую многострочную программу.

Начинаем расти

Назовите вслух все части речи, использованные в этом фрагменте:

```
require 'net/http'
Net::HTTP.start( 'www.ruby-lang.org', 80 ) do |http|
print( http.get( '/en/LICENSE.txt' ).body )
end
```

Первая строка представляет собой вызов метода. В ней используется метод с именем require. Методу передается строка 'net/http'. Можете рассматривать ее как отдельное предложение. Мы приказали Ruby загрузить вспомогательный код, а именно библиотеку Net::HTTP.

Следующие три строки связаны друг с другом. Константа Net::HTTP ссылается на библиотеку, загруженную в предыдущей строке. Мы используем метод start из этой библиотеки. При вызове методу передается строка 'www.ruby-lang.org' и число 80.

Слово do открывает блок, имеющий одну переменную блока http. Внутри блока вызывается метод print. Что же он выводит? Для переменной http вызывается метод get, которому передается строка с путем '/en/LICENSE.txt'. Обратите внимание: сразу же после get по цепочке вызывается другой метод с именем body. Наконец, ключевое слово end завершает блок.

Ну как? Просто из любопытства — сможете ли вы предположить, что делает этот пример? Надеюсь, вы начали улавливать некоторые закономерности в Ruby. Если нет — энергично потрясите головой, пока эти примеры находятся у вас в мозгу, и код сам собой рассыплется на понятные фрагменты.

Например, на практике часто применяется следующий стереотип:

```
переменная . метод ( аргументы метода )
```

Мы видим его внутри блока:

```
http.get( '/en/LICENSE.txt' )
```

В этом примере мы используем Ruby для получения веб-страницы. Вероятно, ранее сокращение HTTP уже встречалось вам при работе с браузером. HTTP означает «Hypertext Transfer Protocol», то есть «протокол пересылки гипертекста»: этот протокол используется для пересылки веб-страниц по Интернету. Представьте курьера, который разъезжает по Интернету и развозит нам веб-страницы. У него на кепке вышиты буквы HTTP.

Переменная http — это тот самый курьер, а метод содержит сообщение для него. Мы запрашиваем веб-страницу, которая называется /en/LICENSE.txt.

Обратите внимание на цепочку вызовов:

```
http.get( '/en/LICENSE.txt' ).body
```

Поскольку мы получаем веб-страницу от курьера http, эту строку можно мысленно прочитать так:

```
веб страница .body
```

Теперь вернемся к коду

```
print( http.get( '/en/LICENSE.txt' ).body )
```

Вызов get дает нам веб-страницу. Мы отправляем странице сообщение body и получаем весь HTML-код в виде строки. Полученная строка выводится на экран. Обратите внимание на цепочечный вызов методов. В следующей главе мы проанализируем всевозможные стереотипные схемы в Ruby; уверяю, это очень интересно. Итак, что же делает этот код? Он выводит на экран HTML-код домашней страницы Ruby; при этом он пользуется услугами курьера, обслуживающего Web.

Наша поездка подходит к концу

Похоже, у нас возникла проблема. Мне кажется, что вам начал нравиться этот процесс. А ведь мы еще даже не дошли до главы, где я буду описывать процесс разбора XML на примере детских считалок!

Если вам действительно это нравится — дело плохо. Через пару глав вы начнете писать свои собственные программы на Ruby. В частности, вы начнете писать собственную программу ведения блога, собственную сеть обмена файлами (в стиле BitTorrent), а также программу, оповещающую вас о приходе электронной почты. За ними последует совершенно потрясающий сценарий: программа, обшаривающая весь Интернет в поисках MIDI-файлов!

Знайте (а вы должны это знать!), что это превратится в одержимость. Для начала вы забудете вывести на прогулку свою собаку. Она будет печально стоять у двери и вертеть головой, пока ваши глаза будут жадно поглощать код, а пальцы — отправлять сообщения компьютеру.

Из-за вашего пренебрежения дом придет в запустение. Вороха распечаток поднимутся до вентиляции. Батареи перестанут греть. На кухне взгромоздятся мусорные горы: поспешно заказанные обеды с доставкой на дом и ненужная почта, от которой вам было лень избавиться. Воздух пропитается запахом вашего немытого тела. У потолка будет расти мох, вода начнет застаиваться в трубах, в дом начнут забегать дикие животные, а сквозь пол прорастут деревья.

Но ваш компьютер будет в полном порядке. И вы сами будете питать его своими знаниями. За бесчисленные часы, проведенные за компьютером, ваше сознание отчасти сольется с процессором, и вы начнете прирастать к компьютеру. Ваши руки напрямую подключатся к портам, а глаза будут воспринимать видео прямо с разъема DVI-24. Легкие будут находиться над процессором, охлаждая его дыханием.

И когда заросли будут готовы поглотить вас вместе с компьютером, вы закончите свою работу. Вы и ваш компьютер вместе запустите свой замечательный сценарий Ruby, результат вашей одержимости. И сценарий запустит механические пилы, которые истребят лишнюю растительность, и очаги, которые согреют ваш дом и обеспечат нормальную циркуляцию воздуха. Нанороботы-строители, хлынувшие из строк кода, восстановят ваше жилище, сменят кафель, обновят, отполируют и дезинфицируют обстановку. Могучие андроиды превратят разваливающийся сарай в прочное, мощное строение. Они воздвигнут колонны и вырежут из камня статуи. Вы станете полноправным властителем этого роскошного особняка и прилегающих к нему гор и островов.

Короче, я надеюсь, у вас все будет хорошо. Ну что скажете? Приступим к вашему первому сценарию?

Алфавитный указатель

Лучшие примеры разработки ПО

На титульный лист книги вынесен титульный лист первого полного англоязычного издания «Начал» Евклида, опубликованного в 1571 г.

Интересно заметить, что эту книгу перевел с греческого сэр Генри Биллингсли, ставший лорд-мэром Лондона (вряд ли кто-нибудь из современных политиков обладает познаниями в математике и греческом, необходимыми для выполнения такой работы). В переводе Биллингсли помогал Джон Ди — тот самый Джон Ди, прославленный в художественной литературе (например, в рассказе Г. Лавкрафта «Ужас Данвича»). Кстати сказать, большинство ученых того времени, включая самого Ньютона, принадлежали к обоим мирам — науки и магии. Джон Ди написал замечательное предисловие к книге, в котором утверждалась центральная роль математики во всех науках и искусствах, — тем самым он завоевал расположение всех математиков последующих поколений, включая нашего издателя Гэри Корнелла (Gary Cornell). В частности, в предисловии говорилось:

«Сколь великий соблазн, сколь восхитительное обольщение — наука, главная тема которой так стара, так чиста, так превосходна; которая окружает всех живых существ и так часто используется во всемогущей и непостижимой мудрости Творца».

При всей своей важности перевод не был лишен недостатков. Самый заметный из них — ошибка на титульном листе: Биллингсли приписал «Начала» Евклиду из Мегары. Но в действительности Евклид Александрийский работал в великой библиотеке!

Наша копия титульного листа позаимствована из экземпляра, хранящегося в библиотеке Бэнкрофта Калифорнийского университета в Беркли. Она была отсканирована с разрешением 2400 dpi хорошими людьми из библиотеки Бэнкрофта. С изображения был удален налет, характерный для старых книг, а также внесены другие необходимые усовершенствования; этим занимался Курт Крамс, главный дизайнер Apress, в Adobe Photoshop на Mac G4.

Тема: **Программирование** | Уровень читателя: **эксперт**

ПИТЕР

197198, Санкт-Петербург, а/я 619
тел.: (812) 703-73-74, postbook@piter.com

61093, Харьков-93, а/я 9130
тел.: (057) 712-27-05, piter@kharkov.piter.com

ISBN 5-469-01291-3

9 785469 012917

www.piter.com — вся информация о книгах и веб-магазин