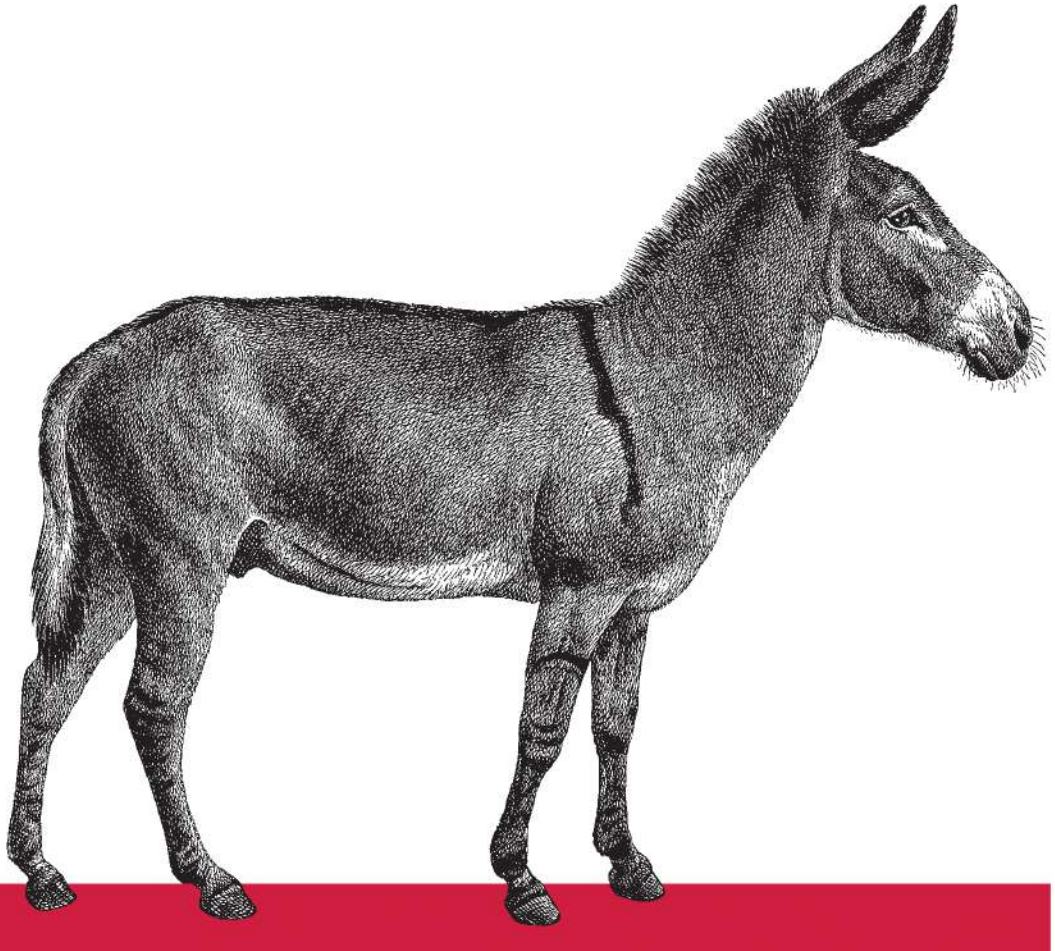
O'REILLY®



ПРОГРАММИРОВАНИЕ БЕЗ ДУРАКОВ

Катрин Пассиг
Йоханнес Яндер

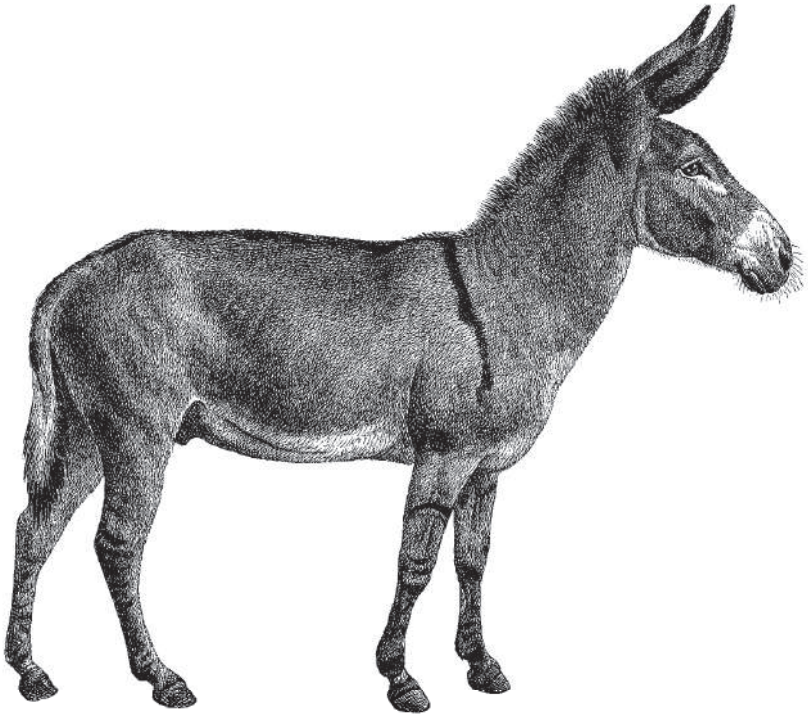
ПИТЕР®

Weniger schlecht programmieren

Kathrin Passig & Johannes Jander

O'REILLY®
Beijing · Cambridge · Farnham · Köln · Sebastopol · Tokyo

Катрин Пассиг
Йоханнес Яндер



ПРОГРАММИРОВАНИЕ БЕЗ ДУРАКОВ

ПИТЕР®
Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону • Самара • Минск
2017

Катрин Пассиг, Йоханнес Яндер

Программирование без дураков

Перевели с немецкого Е. Зазноба, А. Кидрон, Е. Коробейников

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*О. Сивченко*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Н. Рощина*
Художник	*С. Заматевская*
Корректоры	*О. Андриевич, Е. Павлович*
Верстка	*А. Барцевич*

ББК 32.973.23
УДК 004.3

Пассиг Катрин, Яндер Йоханнес

П19 Программирование без дураков. — СПб.: Питер, 2017. — 416 с.: ил.

ISBN 978-5-496-02023-7

Хотите научиться программировать «less wrong»? Тогда эта книга — для вас. Ведь программирование — это во многом коммуникация. Стиль программирования, именование, комментирование, работа с чужим кодом — зачастую соглашения складываются именно там, где строгая регламентация на уровне языка программирования отсутствует. Познакомьтесь с разнообразными традициями, существующими в различных языках программирования, узнайте, как, практически не спотыкаясь, передвигаться по этой пересеченной местности. Со знанием дела и юмором авторы погружаются в обсуждение ложных путей, неверных суждений и ошибок, тем самым значительно облегчая жизнь любому начинающему и бывалому программисту.

6+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ISBN 978-3897215672 нем.
ISBN 978-5-496-02023-7

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), 3, литер А, пом. 7Н.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Подписано в печать 14.12.16. Формат 70x100/16. Бумага офсетная. Усл. п. л. 33,540. Тираж 1000. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор»
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87

Краткое содержание

Оглавление

Предисловие

> Человек знает о существовании неисчислимого множества вещей, о которых он ничего не знает. Количество вещей, о существовании которых человек только подозревает, но больше ничего не знает, служит довольно запутанным примером. Но это вовсе не плохо, поскольку, если человек знает, что он чего-то не знает, он просто может ввести поисковый запрос в Google и... вуаля. Сложнее становится, если человек не знает даже о существовании явлений, о которых он ничего не знает. Незнание такого рода, к сожалению, открывает человеку путь к сильным потрясениям.
>
> *Кай Шрайбер, проект «Riesenmaschine — абсолютно новая Вселенная»*

«Мы все должны стремиться к тому, чтобы стать более компетентными программистами; если у вас нет этого стремления, эта книга не для вас», — пишет во введении к своей книге «Сила кода» Пит Гудлифф. Наша же книга дает тот самый толчок, которого вам как раз не хватает. Или стремления у вас нет, но есть другие приоритеты, а профессиональное саморазвитие находится как минимум на восьмом месте в этом списке. Если же вы чувствуете, что хотели бы избавиться от постоянно возникающих в работе проблем, то эта книга для вас.

Вы хотите целенаправленно совершенствовать навыки программирования, но у вас в этом нет острой потребности, а еще вы не можете найти тот образец хорошего стиля программирования, к которому нужно стремиться. Вы не намерены становиться архитектором ПО или техническим директором фирмы, а просто хотите решать каждодневные проблемы и при этом не желаете дополнительно платить за такую работу.

Возможно, вы сами еще далеки от того, чтобы называть себя программистом. Поскольку программирование уже вовсе не тайное знание особо посвященных специалистов, вы решили немного освоить этот навык для удовлетворения своих амбиций или облегчения ранее незнакомой вам работы. Вы написали какую-то программу и в определенной степени овладели по крайней мере одним языком программирования. Вы находите решения проблем, но чувствуете, что все это определенно могло бы работать лучше. Вы задаетесь вопросом: а как это все происходит у других программистов?

Или вы, даже несмотря на небольшой опыт, считаете себя способным программистом, который только иногда сталкивается с небольшими трудностями. Это стадия неосознанной некомпетентности. Когда мы проводили опрос перед написанием этой книги, мы выяснили, что такое состояние может длиться от 10 до 15 лет.

> Из всех 19 лет, которые я занимался программированием, 13 лет я определенно программировал плохо. Сначала возникали небольшие проблемы, которые заставляли меня размышлять. Когда начинаешь создавать что-то для других и ничего не получается или ты выбираешь более короткий путь, нередко случается так, что эти продукты слишком скоро перестают работать. Поскольку люди используют программное обеспечение по-другому, поскольку они не замечают ошибок и не избегают их автоматически. В связи с этим мне приходилось выслушивать язвительную критику своих клиентов. Я не понимал тогда, что я плохой программист. Я думал, что моих умений вполне достаточно. Но стресс при общении с клиентами сказывался на моих нервах. Если проект становился неудачным, для меня это было очередным провалом, и однажды я подумал: «Так не может продолжаться. Как сделать это лучше?» После этого дела пошли в гору.
>
> *Лукас Хартманн, разработчик ПО*

В психологии развития с 1970-х годов существует модель развития понимания и способностей в комплексных областях знаний, так называемые четыре стадии компетентности[1]. Даже если такая классификация с научной точки зрения обоснованна лишь в определенной степени, она отлично иллюстрирует процесс профессионального становления программиста от новичка до опытного специалиста.

Первая стадия представляет собой неосознанную некомпетентность, когда программист еще не знаком с понятиями области знаний, а также не осознает, что у него есть пробелы в знаниях. За неосознанной некомпетентностью следует осознанная некомпетентность: у специалиста все еще возникают большие трудности при решении проблем, однако он их уже осознает. Вполне возможно, что вы уже прошли данный этап, иначе у вас не было бы причин для чтения этой книги. Возможно, над вашим кодом смеялись другие, а может, вы однажды случайно стерли базу данных, важную для проекта, или, как вариант, вы просто по природе скромный человек. **После этих этапов следует стадия осознанной компетентности: уже под силу работа с ключевыми элементами системы, но нужно еще уделить внимание правильному исполнению задач.** Наконец, финальным этапом является стадия неосознанной компетентности. Все происходит само по себе. После прочтения этой книги вы должны комфортно прочувствовать на себе такое состояние и изредка ощущать приливы осознанной компетентности.

Втайне, возможно, вы чувствуете себя немного медлительным, несообразительным и технически менее подкованным, чем другие. В реальности же мир полон квалифицированных программистов, которым сложно выполнить простейшие расчеты в уме. Плохой программист плох не потому, что у него пока мало опыта, его IQ недостаточно высок, у него плохая память, он самоучка или поздно решил стать программистом. Все эти факторы либо не играют большой роли, либо даже могут сыграть специалисту на руку, если их правильно трактовать (об этом поговорим позже). Плохой программист плох лишь по той причине, что он плохо пишет код.

---

[1] См. en.wikipedia.org/wiki/Four_stages_of_competence.

Почему нужно столько времени, чтобы поумнеть?

Возможно, вы хорошо знаете о проблемах, которые есть в вашем коде. Возможно, у вас уже даже сложилось представление о том, что нужно или не нужно делать в будущем, чтобы стать хорошим программистом. «Нет ничего страшного в том, что вы плохой или средненький программист, — пишет Стив Макконнелл в книге *Code Complete*. — Вопрос заключается лишь в том, как долго программист может быть плохим или средненьким, не осознавая при этом, что можно делать лучше»[1]. Почему же зачастую так сложно перейти к этапу определения проблем и нахождения способов их устранения?

Основные причины того, что мы находим неправильные способы решения проблем, лежат в плоскости наших человеческих качеств. На первом месте в этом случае стоит консерватизм. Сначала не так уж просто противостоять сильному желанию делать все так же, как и раньше. Мозг должен экономно распределять свои ресурсы и поэтому функционирует таким образом, чтобы не отвергать приемлемое решение проблемы при виде туманной альтернативы на горизонте. А изучение всех актуальных трендов в новых технологиях, языках, методах и фреймворках — слишком затратное по времени мероприятие, которое может заставить позабыть о веселой жизни.

В то время как дети и подростки большую часть времени узнают что-то новое, взрослые часто не совершенствуют полученные знания в течение длительного периода. «Ну да, но ведь всегда же так было», — скажет плохой программист, у которого нет никакого желания даже представить себе, как сделать мир лучше и не тратить целый день на изменение всех строчных букв в тексте на прописные да еще к тому же расставить точки над «Ё». Он это делает не только потому, что хочет использовать проверенное решение проблемы, он еще хочет по вполне объяснимым причинам затратить на размышления над проблемой как можно меньше времени. Если делать что-то, не думая об этом, к цели придется идти довольно долго, хотя с каждым шагом достигаются видимые результаты. Если же сначала думать над проблемой, то ее можно в конце концов решить довольно быстро, но в первые часы, дни, недели размышлений не стоит ожидать видимого результата. Такое нежелание вкладывать собственные ресурсы в самообразование становится постоянным, хотя при этом результаты самым парадоксальным образом оказываются итогом усердной работы. В то же время программист, который долго думает над проблемой, напоминает человека, постоянно сидящего на балконе и устремившего вдумчивый взгляд на облака или целый день листающего вебстраницы.

Некая доля нежелания получать дополнительные знания возникает также из боязни: боязни нового, неизвестного, сложного, боязни профнепригодности. Специалисты, продолжающие использовать в работе устаревший фреймворк, устаревший язык программирования, выглядят в глазах других по меньшей мере недостаточно

---

[1] McConnell S. Code Complete: A Practical Handbook of Software Construction, Second Edition. — Microsoft Press, 2004. — P. 825.

компетентными. В новой области такие специалисты внезапно становятся новичками. По сравнению со страхом умереть от нападения зомби в темной пещере все перечисленные страхи имеют реальные основания, и их достаточно для того, чтобы оградить нас от всего неизведанного.

Проблемы, характерные для плохих программистов

Как и всем остальным, плохим программистам в процессе совершенствования приходится бороться с упомянутыми сложностями. Но им присуща еще пара особых проблем. Зачастую они заранее предъявляют к себе завышенные требования. В начале профессионального пути программист уже недоволен тем, что на мониторе постоянно появляются пустяковые сбои. Возникает проблема «для взрослых», которую новичок непременно хочет решить, например убрать ошибки в биржевых курсах. И, еще не обладая соответствующими знаниями, он пытается решить эту интересную, но чрезвычайно сложную проблему.

Кроме того, в целях безопасности программисты стараются не показывать другим свой код. В связи с этим у них не только не прибавляется мотивации от потенциальной критики или похвалы, но и становится недоступной возможность получения дополнительных знаний, а именно возможность перенять опыт у более квалифицированных специалистов. Зачастую небольшое объяснение от опытного программиста может по своей ценности превзойти недельное чтение литературы, ведь мы не можем найти те понятия, с которыми даже не знакомы. А тот, кто с ними уже знаком, вероятнее всего, будет знать, на каких этапах работы эти понятия смогут облегчить жизнь начинающему специалисту, а также сможет их объяснить, акцентируя внимание на эффективности их использования для решения конкретных проблем.

Еще одним страхом может стать страх перед собственным кодом. Человек признает необходимость новой структуры для своего кода, чтобы тот стал более понятным. Однако программист может опасаться, что его неудобочитаемый код с правильным решением проблемы может стать удобочитаемым, но с ошибочным результатом. В итоге он придерживается правила «Не нужно менять то, что и так работает». Если же система работает не так, как хотелось бы, вместо привычного спокойствия появляется трепет перед написанным кодом.

Наконец, точка зрения типа «Ну, меня это не касается, я вообще программирую только ради удовольствия» тоже не пойдет на пользу саморазвитию. За ней стоит представление о том, что самообразование или улучшение навыка программирования доставляет меньше удовольствия, чем непосредственно работа с кодом. Поэтому такое суждение в корне неверно. Меньше всего удовольствия, наверное, доставляют только моменты размышления над собственными ошибками. В худшем случае за этим следует умственная тренировка навыков применения новых технологий. Время, потраченное на этом этапе, окупается в ходе дальнейшей работы, а именно тогда, когда не приходится ночами плутать в собственном неразборчивом коде.

Топ-7 отговорок плохих программистов

«Мой код не видит никто, кроме меня».

- Уже завтра можно встретить специалиста, который пожелает вместе с вами участвовать в этом прекрасном проекте. Повезет вам только в том случае, если это будет некто, не владеющий немецким языком, а код будет содержать имена переменных и комментарии на немецком языке.
- Из суждения «Мой код не видит никто, кроме меня» в скором времени получается «Мой код не позволено видеть никому, кроме меня». В результате программист цепляется за ненужные проекты и чувствует себя загруженным, а проекты за длительное время могут и наскучить. После нескольких лет благих намерений всецело посвятить себя написанию кода программист в конце концов все равно вынужден покинуть рабочее место. Недоработанное ПО, которое уже нельзя доработать, умирает бессмысленной смертью.
- Плохо думать, что программист сам должен смотреть свой код, в особенности спустя годы, когда он уже стал опытным разработчиком.

«Никто не пользуется программой, кроме меня».

- Уже совсем скоро программист наверняка и думать об этом перестанет, но еще он может забыть внедрить в систему необходимые функции безопасности.
- Даже вы сами скоро станете другим человеком, который уже и не вспомнит, о чем нужно помнить при использовании этой программы.

«Я все сделаю как следует, только позже».

- Один из основных законов вселенной гласит: временные меры действуют дольше других. Доказано на практике: код, написанный небрежно и необдуманно, будет применяться программистом еще очень долгое время.
- В 90 % случаев это «позже» вовсе не наступает, поскольку, во-первых, появляются новые проекты, требующие внимания, а во-вторых, ни у кого не возникает желания притрагиваться вновь к небрежно написанному коду. Наконец, в-третьих, программист пытается скрыть, насколько плох код. Иногда внезапно до наступления этого «позже» приходит новое тысячелетие с неожиданно высокими требованиями к запоминающим устройствам, касающимися работы с датами, и выполнить их быстро довольно сложно.

«Это, вообще-то, очень сложная проблема, ничего не выйдет, если только я не применю восемь вложенных циклов».

- Даже при решении сложнейших проблем можно обойтись без восьми вложенных циклов.
- Лучше даже не думать о создании восьми вложенных циклов как раз при решении сложнейших проблем.

- При более тщательном рассмотрении сложная проблема постепенно преобразуется в набор менее сложных проблем, последние, в свою очередь, состоят из еще менее сложных проблем, которые представляют собой набор обычных задач. Нужно лишь написать большое количество простых функций (а еще лучше — использовать то, что уже написано другими) — и сложнейшая проблема решена.

«Я просто понимаю, что определенные данные нельзя вводить».

- Нет, этого делать и не стоит. Когда вокруг суматоха, вас внезапно попросили представить свою новинку мировой прессе или вы смогли произвести отладку сложнейшей проблемы, вы можете ненароком ввести те данные, которые приведут к краху вашей базы данных.
- Другие программисты, которые только пришли на ваш проект (см. выше), могут и не подозревать, что ввод каких-то данных запрещен.

«Я уже думаю над тем, когда стоит комментировать код».

Это твердое убеждение уже не раз становилось предпосылкой для романтических ночей за монитором, отведенных для отладки, и это происходило еще до вашего рождения. Вы думаете, у вас все будет по-другому? Довольно сложно думать о проекте, если не замечать лежащие где-то неподалеку невзорвавшиеся гранаты.

«Это всего-то маленький проект».

- У любого проекта множество измерений. Код, который может решить узко очерченную проблему, часто может использоваться на протяжении многих лет. Таким образом, проект может быть небольшим, но долгосрочным.
- В самом начале любой проект мал. Если вы будете думать, что это всего-то маленький проект, он никогда не разовьется в нечто большее. Такие проекты гибнут задолго до того, как могут стать крупными, из-за малого угла обзора.

Несколько лет спустя

Что же нужно для того, чтобы быть не таким уж плохим программистом? Да немного, по сути. Любознательность помогает, равно как и спокойное осознание своей полной неосведомленности, к примеру, в случаях, когда нужно решиться попросить совета у других. Следует быть готовым терпеливо не понимать некоторых вещей. Нужно читать, а не просматривать тексты и понимать, что сразу невозможно воспроизвести в уме функционирующую модель реальности. Не нужно злиться, если после двух часов возни с новым понятием никак не становится понятно, о чем же все-таки пишет автор книги «C++ за ночь».

Обучение — затяжной процесс. Он длится многие годы, а некоторые утверждают, что вообще всю жизнь. Важные процедуры самообучения могут и немного ускоряться, однако на них все равно нужно выделять время. Изобретатель языка программи-

рования Erlang Джо Армстронг поясняет в книге *Coders at Work*[1], как он на протяжении всей своей жизни становился умнее: сначала он должен был написать программу, чтобы удостовериться в том, что она работает, и только 20 лет спустя он мог уже все это мысленно представить. Однако он все равно отводит время на самообучение — тестирование длится один год, рефлексия — столько же. Отличается одно от другого лишь тем, что в последнем случае не нужно так долго сидеть за компьютером[2].

Прочитав эту книгу, вы не станете сразу умнее. И не станете хорошим программистом. Дойдя до конца книги, вы можете лишь стать чуть лучшим специалистом, чем сейчас. Такой программист знает: за то, что он сегодня натворил, завтра ему придется расплачиваться. Он знает, в каких местах может ненадолго облегчить себе жизнь, но при этом надолго заварить кашу. Он внимательно относится к своим ошибкам. Он не боится показать свой код другим и выслушать критику. Он начинает сам нести ответственность за свои поступки. Он не говорит: «API устарел», «PHP ничем не лучше других», «Проблемы из-за ошибок в библиотеке, а с этим я ничего не поделаю» или «Оно должно было работать быстрее». Не такой уж плохой программист чувствует себя ответственным даже за те проблемы, которые от него не зависят, и пытается всеми силами их устранить. Благодаря большей осведомленности он не делает ошибок.

Ну или по крайней мере делает их не так часто, как раньше.

Что же вас ждет на следующих 393 страницах

Эта книга содержит много текста и мало примеров кода. Вы наверняка знаете о том, что конкретные рекомендации по написанию более качественного кода можно найти в самой рабочей среде. Однако что-то вам до сих пор мешало найти такие рекомендации и уделить им должное внимание. Наша книга описывает причины такой боязни самообразования, а также призвана смягчить сложнейшие проблемы таким образом, чтобы их решение требовало наименьших усилий.

Часть I «~~Привет, миф!~~ Привет, мир!» посвящена ключевой проблеме — хрупкому балансу между заниженной и завышенной самооценкой и процессу перехода от неосознанной некомпетентности к осознанной. В части II «Программировать и понимать» речь пойдет о том, каким образом самому и другим можно прийти к пониманию цели написания кода. В части III «Работа над ошибками» мы поговорим о том, что значит допустить ошибку и какие проблемы при этом могут возникнуть. Как можно тем же умом, что допустил ошибку, увидеть ее, устранить и, как вариант, избегать в будущем? В части IV «Выбор средств» описывается инструментарий программиста. Что не обязательно делать самому, что категорически не стоит делать самому? Возможно, после прочтения книги вы

---

1 *Сейбел П.* Кодеры за работой. Размышления о ремесле программиста. — М.: Символ-Плюс, 2011. — 544 с. (*Seibel P.* Coders at Work: Reflections on the Craft of Programming. — Apress, 2009)

2 *Сейбел П.* Кодеры за работой. — М.: Символ-Плюс, 2011. — С. 200.

и не станете заниматься объектно-ориентированным программированием или не захотите применять определенную среду разработки, но по крайней мере вы будете знать, для чего были придуманы эти инструменты и какие процессы могут проходить быстрее (на тот случай, если вы все же однажды поменяете свою точку зрения).

В конце каждой главы и в конце книги мы предлагаем вам источники для дополнительного чтения. Если по той или иной теме написана хорошая статья в немецкой «Википедии», мы ее обязательно упомянем, если нет, можем указать ссылку на статью на английском. Если в мире действительно столько плохих программистов, сколько мы насчитали в результате нашего маркетингового исследования, эта книга может разойтись хорошим тиражом. Вы очень поможете нам и будущим читателям, если сообщите о найденных ошибках, сделаете критические замечания, предложите идеи для совершенствования книги, прислав их по адресу: wenigerschlechtprogrammieren@kulturindustrie.com.

ЧАСТЬ I

~~Привет, миф!~~ Привет, мир!

1 Туда ли я попал?

Чтобы узнать, нужна ли вам эта книга, потратьте немного времени и просмотрите вопросы, перечисленные далее. Отвечать нужно честно и без долгих раздумий. Если вы вообще не понимаете, о чем идет речь в каком-то варианте ответа, не думайте и переходите к другому вопросу.

Я пишу программу:

а) в Блокноте;

б) в браузере;

в) в другом источнике.

Если происходит сбой:

а) я публикую описание сбоя под названием «Помогите!!!» и привожу точный список всего используемого «железа» на подходящем форуме;

б) я ввожу много строк `print`, которые выводят мне содержимое переменных;

в) я занимаюсь отладкой на GDB.

Для проверки версии я использую:

а) ничего не использую. Если вдруг я что-то случайно сотру, мне придется заново все писать. Поэтому я тщательно этого избегаю;

б) SVN;

в) Git или mercurial.

Я комментирую код:

а) никогда не комментирую, потому что мне не хочется долго сидеть за компьютером;

б) никогда не комментирую, потому что думаю, что мой код понятен и без комментариев;

в) никогда не комментирую, потому что мой код не требует разъяснений.

Если мне нужен XML-парсер:

а) я трачу выходные на его написание, как бы сложно это ни было;

б) мне он не нужен;

в) я читаю статьи о SAX- и DOM-парсерах в «Википедии», просматриваю разные библиотеки и их узлы для используемого мной языка программирования, взвешиваю их плюсы и минусы и ищу активные сообщества, связанные с этой темой.

Для валидации почтового ящика:

а) я быстро ввожу две строки, которые проверяю на собственном адресе электронной почты;

б) я проверяю, включен ли в адрес знак @;

в) я ищу в поисковике регулярное выражение (Regular Expression), которое уже было опробовано надежными проектами.

Мой код и его распространение:

а) я держу код в секрете тщательнее, чем государственную тайну;

б) если мой код увидит кто-то, кому уже исполнилось полгода, мне будет немного стыдно;

в) мой код входит в ядро Linux.

Я тестирую код:

а) вообще не занимаюсь этим. Если что-то не работает, рано или поздно я замечаю это;

б) после каждого изменения кода;

в) вообще не занимаюсь этим. После каждого изменения код тестируется на сбои автоматизированными юнит-тестами.

Как выглядит международный стандарт формата даты и времени:

а) Д.М.ГГ, а как еще;

б) количество секунд, прошедших с полуночи 1 января 1970 года, — Всемирное координируемое время, при котором секунды не считаются, а сохраняются в 64-битном операторе;

в) ISO 8601.

Оптимизация:

а) меня не волнует;

б) применяется мной до тех пор, пока программа не начнет выполняться за 43,3485 тактового цикла на моем компьютере 2,4 ГГц Core 2 Duo с 256 Мбайт кэша второго уровня;

в) меня не волнует, если на взаимодействие с пользователем (User Experience) время ожидания никак не влияет.

Другие программисты:

а) лучше меня;

б) хуже меня;

в) бывает по-разному.

Я совершенствую свой код:

а) никогда не совершенствую, но всегда рассчитываю переписать его, чтобы он был более качественным;

б) шаг за шагом, когда у меня есть свободное время;

в) шаг за шагом, даже когда у меня на это нет времени.

Теперь подсчитайте, сколько каких у вас ответов. Если больше ответов *«а»* и *«б»*, то эта книга определенно для вас. Если же у вас больше половины ответов *«в»*, не смейтесь, пожалуйста, над нами и продолжайте дальше программировать драйвер в ядре Linux. Или чем вы там еще занимаетесь.

2 От гордыни к смирению

> Мне регулярно приходится гуглить синтаксис языка, который я использую каждый день на протяжении 10 лет. #coderconfessions.
>
> *@HackerNewsOnion, Twitter, 10 июля 2013 года*

В последние годы в околокомпьютерных кругах часто вспоминают об эффекте Даннинга — Крюгера[1], согласно которому именно непрофессионалам крайне свойственна завышенная оценка своих способностей. Результаты исследований указывают на то, что в реальности все по-другому и гораздо проще: люди в целом плохо оценивают свою компетентность, в основном лишь до некоторой степени правильно.

Программисты с небольшим опытом работы колеблются между завышенной самооценкой и мыслью о том, что они совсем глупы для профессии. Когда в ходе планирования нового проекта к ним приходит вдохновение, они нередко высоко оценивают собственные способности и в то же время у них не возникает даже представления о том, насколько длителен процесс написания исправного кода. Осознание горькой правды приводит либо к отчаянию (в проектах с дедлайном), либо к нежеланию продолжать что-то делать (в проектах для души).

У завышенной самооценки есть предпосылки. Начинающие разработчики постоянно чему-то учатся, и это льстит их эго. Если изобразить процесс обучения в виде графика, то в первые 12 месяцев обучения чему-то новому прямая будет так круто подниматься вверх, что от успехов может закружиться голова. Однако очень сложно распознать то состояние, когда, несмотря на долгое и упорное самообразование, программист все равно может только кое-как барахтаться в лягушатнике. Мир программирования для новичка полон «неведомой безызвестности», говоря языком Дональда Рамсфелда, то есть полон пробелов в знаниях, которые можно вообще не осознавать.

Другая причина завышенной самооценки заключается в том, что неопытные программисты склонны завершать только 80 % заданий проекта и прекращают над ним работать, как только им становятся неинтересны поставленные задачи. Согласно принципу Парето[2] в эти оставшиеся 20 % как раз заложено 80 % работы. Можно создать образцовое приложение, даже обладая небольшими знаниями. Проблемы,

---

1 https://ru.wikipedia.org/wiki/Эффект_Даннинга_—_Крюгера.

2 https://ru.wikipedia.org/wiki/Закон_Парето.

вызванные незнанием или неудачными решениями, показывают свое истинное лицо не в начале проекта, а лишь в ходе работы над последними 20 % задач.

Переоценка собственных способностей или недооценка задачи могут обеспечить и некоторые преимущества. Если бы каждый специалист в самом начале своей карьеры мог точно определить степень своей некомпетентности, человечество все еще жило бы в пещерах («Небоскребы? Мы же ничего не знаем об этом!»). Даже в отдельных проектах иногда полезно ошибаться в оценках. Когда Дональд Кнут, недовольный текстом второго тома своей главной работы «Искусство компьютерного программирования», сам решил в 1977 году написать хорошую программу, на это ему понадобилось около полугода. В результате только почти через 12 лет была выпущена программа TeX. Докторант Джордж Бернард Дантциг, обучаясь на курсе статистики в Университете Калифорнии в 1939 году, слишком поздно обнаружил два недочета в выполненном домашнем задании. Через несколько дней он извинился перед профессором за затянутое выполнение задания: мол, они были сложнее, чем казалось раньше. Однако домашнее задание здесь было ни при чем, Дантцигу совершенно случайно удалось вывести ранее неизвестную в статистике теорему. Таким образом, в некоторых случаях ошибочные решения могут пойти только на пользу, поскольку они помогут не бояться сложности и запутанности задания.

Отличная новость! У других все происходит точно так же. Даже опытные программисты забывчивы, бывают рассеянны, отлынивают от работы и считают написанный ими код самым ужасным в мире. Существует лишь несколько проблем, свойственных исключительно неопытным или торопливым программистам.

Трудноподдерживаемый код

Зачастую код пишут, не принимая в расчет его поддерживаемость (на том же языке либо на других). Это объясняется в первую очередь недостатком профессионального опыта у программиста. Только если изначально самому попытаться разобраться с непроходимыми джунглями кода собственного производства, можно в дальнейшем при написании кода думать о тех, кто его будет читать.

Выбор неэффективных инструментов

Тот, кто знаком только с одной половиной инструментов языка программирования или определенной методологии, будет использовать эти знания при выполнении всех задач, над которыми работает. Часто проблема заключается не в полном незнании, а скорее в недостаточном доверии к другим методам работы.

Переоценка своих способностей

Даже хорошие программисты часто переоценивают эффективность своей работы. С одной стороны, это объясняется тем, что они, как и остальные люди, не могут реально оценить, сколько будет длиться проект. С другой стороны, зачастую постановка конкретной задачи происходит уже в ходе работы. Плохие программисты в данном аспекте ничем не отличаются, разве что у них дела обстоят в три раза хуже.

Нехватка знаний

Для неопытного программиста ново каждое понятие на его профессиональном пути. К тому же в сфере разработки ПО есть довольно много идей, которые постоянно находят применение в новых областях. Опытные программисты могут распознать идеи такого рода и тем самым облегчают себе работу при использовании знакомых понятий в новых контекстах.

Слабые стороны могут быть сильными

Создатель языка Perl Ларри Уолл в своем основном труде «Программирование на языке Perl» характеризует лень, неусидчивость и завышенную самооценку как важные характеристики программиста. Лень нужна, потому что она мотивирует программистов беречь как можно больше сил. Так они смогут написать оптимальную для работы программу, основательно зафиксировать все, что они написали, и составить список часто задаваемых вопросов, чтобы потом не отвечать на их огромное количество. Неусидчивость позволяет программистам писать ПО, которое не только удовлетворяет их запросы, но и может их предсказывать. А завышенная самооценка позволяет программисту творить чудеса. Более того, другие недостатки разработчиков могут стать их преимуществами, если только их правильно использовать.

Глупость

«Нередко тот, кто пишет самый жуткий спагетти-код, как раз и способен удержать его в голове целиком. Ведь только поэтому он так и пишет», — полагает Питер Сейбел[1]. Не очень способный программист попытается найти наиболее простое решение и, вероятнее всего, с его помощью написать код, который не поймут, не прочитают и не усовершенствуют другие.

Некомпетентность

Уверенное владение языком программирования или абстрактными понятиями — штука хорошая. Но как только приходится изменять парадигму — так случилось при появлении объектно-ориентированного программирования, — многие из считавшихся компетентными программистов страдают как раз из-за своих имеющихся знаний. Ничего же не подозревающие индивиды в такой ситуации адаптируются намного легче, поскольку они могут идти навстречу новому без лишнего груза.

Забывчивость

Дуглас Крокфорд, одна из ключевых фигур в создании JavaScript, так ответил в одном интервью на вопрос о том, является ли программирование сферой профессиональной деятельности исключительно для молодежи: «У меня все прекрасно получается, может быть, даже лучше, чем раньше, потому что я научился не зависеть

---

[1] *Сейбел П.* Кодеры за работой. — М.: Символ-Плюс, 2011. — С. 273.

так сильно от собственной памяти. Я теперь лучше документирую свой код, так как уже не столь уверен в том, что смогу вспомнить через неделю, почему я что-то сделал»[1]. Кроме того, программист, который постоянно забывает имена и синтаксис простейших функций, более заинтересован в освоении новой среды разработки или как минимум редактора с умным автодополнением кода.

Слабая выносливость

В определенных кругах считается подвигом программировать ночь напролет. Однако высока вероятность того, что на следующий день код, написанный в таких условиях, окажется бесполезен. Если можно себе позволить работать без мучительных дедлайнов, устанавливаемых начальниками, можно и не принуждать себя к работе над кодом. Зачастую хорошая мысль, пришедшая в голову во время отдыха у реки, может заменить целые недели с немыслимым графиком работы.

Прокрастинация

Именно плохим программистам полезно откладывать улучшение кода на максимально возможный срок. Иногда откладывание работы только играет на руку. Кроме того, рабочие инструменты, языки программирования, кодовые библиотеки и фреймворки со временем становятся только лучше. Чем больше проходит времени, тем проще становится решение, поскольку другие, опытные программисты к тому моменту уже выполнят часть работы. Довольно часто задача может и подождать, пока проблема не будет устранена в открытом проекте. Активное участие в проекте такого рода только ускорит этот процесс.

Отвращение к собственному коду

Ненавидеть собственный код естественно и хорошо. За хороший код всегда цепляются с необоснованным рвением. Плохой код всегда осознанно отправляют в мусорную корзину, как только пользователи захотят изменений или появится новый способ решения проблемы. Те, кто считает свой код совершенным, ничему новому не учатся. Совместная работа с другими программистами всегда проходит продуктивнее, если все участники проекта критически относятся к собственному коду. Чрезмерное внимание к качеству кода также может отнять много энергии, тогда как она намного нужнее для иного — осмысления целей и результатов работы. Код, в конце концов, всего лишь способ решения задачи и не является самоцелью.

Отсутствие целеустремленности

Сносный код иногда может приносить радость — если не его пользователям, то хотя бы создателям. При исследовании феномена счастья выделяют две категории людей: максималистов, которые всегда стремятся к совершенству, и конформистов, которым нужно не так много, чтобы быть счастливыми. Максималисты стремятся к высоким результатам, но конформисты чувствуют себя более счастливыми.

---

[1] Там же. — С. 114.

Медлительность

Медлительность, направленная в нужное русло, может стать вашим рабочим инструментом. Приведем пример: глобальные переменные нужно использовать только там, где без них действительно не обойтись, то есть весьма редко[1]. Но, поскольку такие данные, доступ к которым требуется из нескольких мест, довольно удобно поместить в одну глобальную переменную, начинающим свойственно слишком часто использовать эту возможность. Чтобы затем избавиться от такой глобально заданной переменной, нужно приложить много усилий. Можно же, напротив, свою медлительность использовать во благо, основательно задав все локальные переменные, которые позже можно преобразовать в глобальные, если все пойдет по плану. Тогда такая медлительность становится полезной и не позволяет программисту испортить код.

Основная проблема, которую представляет зацикленность на собственных слабостях, заключается в том, что их можно использовать в качестве оправданий: «Мне не нужно даже пытаться улучшить это, потому что я не могу. Я не такой сообразительный». Такое отношение не только непродуктивно, но и в корне неверно. Большинство собственных слабостей и неумений, если обратить на них более пристальное внимание, проявляются не так уж часто, и их было бы разумнее воспринимать как данность для характера человека. Среди ваших знакомых, наверно, существует суперпрограммист, который распознает ошибки в коде с первого взгляда, владеет системой обратной польской записи и держит в голове все тонкости синтаксиса 12 любимых языков программирования. Но существование такого суперпрограммиста лишь подтверждает истину о том, что есть и исключения из всеобщей несовершенной посредственности.

Истина не всегда кроется в сложном

Код, написанный самостоятельно, часто похож на текст детской книги: «Это собака. Собака играет в саду. А сейчас собака лает». Но это не повод для отчаяния. Хорошего программиста выделяет не способность написать самый запутанный код. Утонченные и здравые идеи могут встретиться и в незатейливом коде, и, наоборот, за заумным кодом может скрываться плохо продуманная концепция[2].

В авиастроении и других технических отраслях существует одно широко известное правило: «Делай это проще, тупица». С помощью этого принципа конструирования изначально хотели добиться того, чтобы один механик мог починить американский бомбардировщик за две минуты с помощью швейцарского карманного ножа. Если применить эту формулу в сфере разработки ПО, можно смело утверждать, что формулировки «для детского сада» не только приемлемы, но и предпочтительны, поскольку они предельно понятны и легко читаются.

---

[1] Недостатки глобальных переменных приводятся в статье «Википедии» по адресу en.wikipedia.org/wiki/Global_variables.

[2] Особый случай — плохой код, скрывающий плохое понятие: «Иными словами, залог успеха Корпорации на просторах Галактики: существенные недостатки в проектировании обычно скрываются за поверхностными недостатками проектирования» (*Дуглас А.* Всего хорошего, и спасибо за рыбу. — 1984).

Зачастую задачи, с которыми сталкивается в работе программист, не так сложны, как кажется на первый взгляд. Берни Козелл, один из программистов Arpanet — системы-прародителя Интернета, утверждает: «Я выработал пару правил, которые всегда стремлюсь привить людям, особенно недавним выпускникам колледжей, думающим, что они знают о программировании все. Первое заключается в том, что есть совсем немного непреодолимо сложных программ. Если вы смотрите на участок кода и он кажется вам сложным — настолько, что вам даже не понять, для чего он предназначен, — то почти всегда это значит, что вы просто мало над ним думали. И не стоит засучив рукава пытаться прямо сейчас его исправить — лучше вернуться на шаг назад и попробовать понять еще раз. Когда вам это удастся, вы увидите, что на самом деле все гораздо проще. [...] Я достаточно давно в этой профессии, чтобы понимать: в ней есть сложности. Но их очень немного. И чем напряженнее человек размышляет над проблемой, тем проще она становится, и в конце концов понимаешь, как на самом деле просто запрограммировать ее правильно»[1].

Нам нужно нормально относиться к своим ошибкам и ошибкам других. Все может оставаться неверным, и обычно это так и есть (правило, которое нужно зарубить себе на носу при чтении комментария к коду). Совершенно непродуктивно из-за этого считать себя идиотом. Младенец, который учится говорить, не покажет вида, если скажет: «Я бежу». Ребенок и его родители понимают, что он имел в виду, и на этом основаны достижения эволюции и индивидуума. Тот, кто может пользоваться каким-либо языком программирования, чтобы только сказать: «Привет, мир», уже продвинулся довольно далеко, и ему не нужно отчаиваться, прочитав записи блогов, в которых рассказывается о «крутых программистах». Вам не хватает лишь стремления сделать дело хоть в какой-то степени хорошо. Нужно распознавать все лишнее и неверное и быть готовым изменить это — может, и не сразу, но уж точно хоть когда-нибудь.

---

[1] *Сейбел П.* Кодеры за работой. — М.: Символ-Плюс, 2011. — С. 471–472.

ЧАСТЬ II

Программировать и понимать

3 Ты такой же, как все

> Любой дурак может написать код, понятный компьютеру. Только хороший программист способен написать код, понятный человеку.
>
> *Мартин Фаулер. Рефакторинг*

Любой язык программирования в первую очередь представляет собой искусственный язык коммуникации между людьми и машинами. В отличие от естественных языков, обеспечивающих коммуникацию между людьми, в языках программирования все абсолютно однозначно: значение определенной языковой конструкции нельзя интерпретировать по-разному. Такая особенность позволяет машинам понимать этот язык.

То, что в языках программирования не допускается многозначность, вовсе не означает, что в них все предельно ясно, ведь у них есть и вторая важная функция: они являются средством коммуникации для программиста, изначально написавшего программу, и программиста, который позже будет читать этот код. Поскольку люди не обладают логикой компьютера, может случиться так, что программист пишет не то, что он имеет в виду, а другой программист неправильно поймет текст, написанный первым.

Ситуация становится неприятной тогда, когда этот другой программист из-за недопонимания изменяет код или неверно к нему обращается, потому что в этом случае, скорее всего, что-то пойдет не так. Во многих случаях другой программист — тот же, что и первый, просто на пару недель или месяцев старше.

Такое недопонимание приводит к тратам времени и, если их вовремя не выявить и не устранить, — к сбоям и неверным результатам. Хорошие программисты присваивают задаче «устранения недопонимания» самый высокий приоритет.

Каждое из наших суждений, будь то в беседе с другими, при написании книги или кода, содержит лишь часть информации, необходимой для понимания. Другая часть остается в голове — непроизнесенной. Таким образом, то, что мы первоначально говорим, в основном не до конца или вовсе не понятно. Мы замечаем это редко, поскольку мало кто потрудится переспросить. Наш собеседник не расслышал нас, потому что сам в это время придумывал бредни, которыми он ответит, а у читателей не всегда есть возможность следить за ходом мыслей автора до тех пор, пока он наконец вразумительно не изъяснится. Отсутствие обратной связи служит непоследней причиной того, почему мы обычно склоняемся к преувели-

чению способности читателей, слушателей и компьютеров читать наши мысли. «Но об этом же вот здесь написано! — восклицаем мы, когда нас переспросят. — Ну я же говорил об этом!» В реальности же мы в лучшем случае могли только подумать об этом.

При написании понятного кода, присвоении имен параметрам и комментировании мы руководствуемся принципами, в чем-то свойственными безопасному сексу: каждый признает его пользу и положительно относится к нему, но все равно редко прибегает на практике — только в случаях крайней необходимости. Основное препятствие заложено в психологии: в момент написания кода мы не нуждаемся в помощи для понимания кода в будущем. Мы знаем, что мы делаем, вся важная информация еще свежа в нашей краткосрочной памяти. Более того, мы же хотим продвигаться в разработке программы, поэтому не удосуживаемся затормозить процесс написанием комментариев.

Автор в таком случае должен встать на место незнакомого программиста, читающего его код впервые, и подумать, что может послужить поводом для выполнения дополнительных действий и, следовательно, не должно фигурировать в коде.

Попытки автора подумать о читабельности кода зачастую зависят от того, какие отношения у него с воображаемыми коллегами, читающими этот код. Эти отношения могут стать более любезными, если руководствоваться принципом буддизма, использованным в «Фантастической четверке»: «Ты такой же, как все, а все такие же, как и ты». Программистами, которым нужна помощь для понимания нашего кода, завтра станем мы сами, когда попытаемся разобраться в нем. Признание того факта, что собственный код очень быстро становится непонятным, так распространено среди программистов, что уже приобрело форму научного закона. Закон программирования Иглсона гласит: «Любой ваш код, который вы не видели шесть или более месяцев, выглядит так, будто его писал кто-то другой». Стоит пояснить, что Иглсон был оптимистом: на практике хватает лишь трех недель.

Поэтому нужно гнать от себя дьявола, нашептывающего вам: «Да ты и так все запомнишь!» В процессе программирования не помешает представить себя на месте главного героя ленты «Помни», который забывает все через несколько минут, если не записывает. Так мы с уважением относимся к нашему рабочему времени, поскольку, если человек однажды потратил уйму времени, чтобы разобраться в вопросе, это вовсе не означает, что он раз и навсегда зарубил себе это на носу. Даже у понимания есть свой срок годности, поэтому можно уже сегодня сберечь завтрашнее время, записав с разъяснениями собственные выводы. Благодаря этому и другие люди будут лучше понимать код, что тоже будет нелишним.

4 Договоренности

> В своей профессиональной деятельности я работал вместе со многими программистами и знаю некоторых совершенно бездарных типов. Просто ужасно некомпетентных. Аж волосы на голове дыбом от одной мысли встают. В таких случаях о рациональном использовании кода и речи быть не могло!
>
> *Лукас Хартманн, разработчик ПО*

Представим ситуацию. Некий программист поработал на первых двух-трех проектах и особо не прибегал к традиционным методикам. Сейчас он смотрит в зеркало и видит в нем классного специалиста с правильными жизненными ориентирами. Любой, кто попытается ему объяснить, что его код написан не по установленным правилам, сразу же будет объявлен педантом и консерватором. Эта вполне ожидаемая и в какой-то степени разумная реакция может появиться у людей, которые только начинают работать в определенной отрасли и хотят поспорить с теми, у кого уже есть некий опыт работы.

Неприятие начинающими специалистами установленных правил имеет рациональное зерно: с одной стороны, за такими условностями часто скрывается жалкая реальность, в которой группы людей активно изолируются от остального мира какими-то негласными законами и методами — это мы так сказали, мы все делаем правильно. Есть и те, кто делает все неправильно, наступает на собственные грабли, программируя код как вздумается. Это желание изолироваться всего лишь одна сторона медали. Но имеются и более веские аргументы в пользу следования установленным в отрасли правилам.

- Будучи новичками, программисты часто игнорируют правила, установленные в языке, вовсе не из-за собственных убеждений, а просто-напросто по незнанию.
- Собственный стиль программирования может быть отточен только тогда, когда программист в совершенстве овладевает языком, с которым работает. До этого момента высока вероятность того, что собственные новые идеи будут хуже, чем установленные правила. Те, кто осмелился поставить под сомнение придуманное давно, имели благие намерения и также внесли значительный вклад в развитие сферы разработки ПО. Но чтобы быть полезным в этой отрасли, нужно:

- в первую очередь понять придуманное давно;
- хотеть что-то не только изменить, но и улучшить. Вряд ли специалист с небольшим опытом думает о таких требованиях.

❍ Когда собственные идеи уже ничем не хуже установленных правил, человек ограничивается личными умозаключениями, неуместными при чтении чужого кода и работе с коллегами.

Для новичков зацикленность на относительных преимуществах собственного стиля форматирования кода оборачивается лишней тратой времени. В других областях знаний можно добиться большей продуктивности, не тратя при этом много времени. Чтобы прагматично решать проблемы, нужно в первые годы работы слепо верить тому, что советуют опытные программисты, даже если это может показаться странным. Те же библиотеки стандартов, то есть собрания всех функций и классов, совмещенные с компилятором или конвертером того или иного языка, зачастую становятся хорошим подспорьем на начальном этапе работы, потому что с этими стандартами считается большинство программистов.

Если же определенное правило вас раздражает и вы хотите что-то сделать по-другому, попробуйте представить, что работаете с множеством напарников в качестве внештатного программиста: вы используете образцы кода и библиотеки, написанные другими, и наверняка в будущем обратитесь за помощью к коллегам. Именно тогда вы и поймете, что придуманные вами правила могут стать основным препятствием для общения с ними.

Учить ли английский

Коротко на этот вопрос можно ответить так:

```
/**
* Kincskereso halalat vezenyli le.
*/
public void meghal()
{
    Game.jatekVege();
    this.mezo.setCellaElem(null);
}
```

А теперь поясним: языки программирования получили свои основные понятия из английского языка. Исключения можно пересчитать по пальцам, и они не имеют большого значения для решения каждодневных задач[1]. Иногда сам язык или используемый фреймворк диктует определенные названия: в Java исходная функция всегда называется `main`, даже если разработчик говорит только по-немецки. Значит ли это, что программист должен присваивать названия и писать комментарии только по-английски?

---

[1] Список можно найти здесь: en.wikipedia.org/wiki/Non-English-based_programming_languages.

В пользу применения родного языка обычно приводятся следующие аргументы.

- Присвоить подходящие названия и написать понятные комментарии на английском языке намного сложнее.
- Успешный результат ставится под сомнение, если принять во внимание уровень владения английским языком, — слабый английский программиста приводит к путанице и недопониманию.
- Если писать по-английски, можно сделать много орфографических ошибок, поскольку неносители языка не всегда с лету вспоминают, как пишутся, к примеру, слова height и width.

Английский язык для кода имеет следующие преимущества.

- Неанглийский язык кода сразу сообщает другим программистам: этот код написан только для себя и больше никого автор кода заинтересовать им не собирался. Это характерно преимущественно для начинающих, то есть свидетельствует еще и о том, что код не лучшего качества.
- Для многих программистов перспектива разбираться в двуязычной абракадабре вовсе не представляется радужной: `if haus[счетчик] instanceof HochHaus`... Самыми непопулярными у программистов являются смешанные названия методов: `getКоличествоКлиентовfromБД()`. Код, написанный на одном языке, читать намного приятнее. И поскольку немецкий не может быть таким языком, остается лишь использовать английский.
- Тем, кто не работает над своим техническим английским, доступны не все интернет-сообщества. Часто решения проблем можно найти лишь на англоязычных форумах с пользователями из разных стран.
- Если нужно будет обратиться за помощью к пользователям на англоязычном форуме, все равно придется переводить неанглийскую часть кода на английский язык перед тем, как опубликовать свою просьбу.
- Аналогичная ситуация может случиться, если захотелось опубликовать собственный код в качестве хорошего или плохого примера для подражания.
- Английский лишним не будет. И не только в программировании.

Идеальным вариантом написания кода является англоязычный код с англоязычными комментариями. Код, написанный на английском языке, но с комментариями на других языках, довольно широко распространен в качестве неприятного компромисса:

```
// ne pas oublier de renseigner ces valeurs
// sinon l'addon ne pourras pas etre utilisée
_resource = "";
```

В данном случае программист, использующий код, не должен о чем-то забыть. Но о чем и в каких ситуациях? Если вы решаетесь на такой вариант, потрудитесь написать сам код так, чтобы он был понятен его читателю и без комментария. Это, в принципе, неплохое решение проблемы (см. главу 6).

Случай Paamayim Nekudotayim

У PHP-программистов может сложиться ситуация, когда они спускаются на землю, получая сообщение системы Parse error: syntax error, unexpected T_PAAMAYIM_NEKUDOTAYIM. Это предзнаменование неминуемого господства инопланетян на Земле — всего-то сообщение о сбое на иврите, которое появилось как следствие разработки в Израиле системы Zend Engine 0.5. Paamayim Nekudotayim переводится как «двойное двоеточие».

Однако это сообщение хорошо своей полной однозначностью. Легче ввести в поисковик это выражение и разъяснить ситуацию, вместо того чтобы пытаться искать double colon. Но лучше даже не пытаться повторить подвиг разработчиков системы Zend.

Нелишним будет заранее договориться с самим собой или другими программистами, какой вариант английского языка использовать, американский или британский. Такой договоренностью пренебрегают не только неопытные программисты, но и, к примеру, основатели «Твиттера», поэтому каждый, кто использует API этого сервиса, должен годами запоминать, когда писать *favorites*, а когда — *favourites*.

Поскольку довольно много программ написано на американском английском, лучше придерживаться этого варианта, хоть это и несправедливо по отношению к британцам с исторической точки зрения.

Учить английский с ZX81

Когда мне было восемь лет или что-то около того, мой отец однажды вечером принес домой компьютер ZX81. Черный громоздкий аппарат с неудобной мембранной клавиатурой, которая тем не менее для меня имела высочайшую ценность, поскольку на кнопках были прописаны основные команды BASIC. Так, даже несмотря на то, что я не имел ни малейшего представления о значении слов print, goto и next, я мог попробовать написать программу со всеми этими командами и посмотреть, что произойдет при ее запуске.

Вскоре мой отец утратил интерес к программированию, а я по этой, по всей вероятности, самой нерациональной методике обучался программированию в свободное время, то и дело наступая на многочисленные грабли. Через пару недель попалась мне на глаза инструкция к этому самому аппарату, написанная на английском языке. Для того чтобы я основательно понял некую взаимосвязь, присущую командам, в частности, for и next, gosub и return, особенно полезными были образцы кода. Когда в шестом классе у нас наконец-то начался английский, в моем словарном запасе уже было полно ключевых понятий языка программирования С. Еще через некоторое время я наконец-таки смог составлять кое-какие предложения типа «My name is Jan. My pen is in my pencil-case. My pencil-case is black».

Ян Бёльше

Камень преткновения

Скобки, отступы, интервалы для программистов то же самое, что для сварливых супругов тюбик с зубной пастой в ванной: в целом неважно, как его выдавливать, ставить его на колпачок или класть, закрывать или оставлять открытым после использования. Но как только рядом появляется человек с другими предпочтениями, возникают проблемы.

В целом правила форматирования кода сводятся к ответам на следующие вопросы.

Где ставить открывающую фигурную скобку — в конце текущей строки или в начале следующей?

```
public boolean hasStableIds() {
    return true;
}
```

Или так:

```
public boolean hasStableIds()
{
    return true;
}
```

Где ставить закрывающую фигурную скобку?

```
public boolean hasStableIds() {
    return true;
}
```

Или так:

```
public boolean hasStableIds() {
    return true; }
```

Отступы нужно делать табуляцией или с помощью пробелов?

Не очень приятный аспект работы с текстовыми данными — это отступы с помощью табуляции. Поскольку знаки табуляции в программировании нередко используют для создания отступов программных блоков, они имеют особое значение. В языке Python отступ даже входит в семантику программы: например, операторы, находящиеся в текстовых блоках с неправильными отступами, могут оказаться за пределами цикла. Поэтому многие специалисты (особенно разработчики на базе Python) попросту отказываются от выравнивания с помощью табуляции и просят коллег вместо нее использовать определенное количество пробелов.

Какой длины должен быть отступ?

Два пробела:

```
public boolean hasStableIds() {
  return true;
}
```

Или четыре:

```
public boolean hasStableIds() {
    return true;
}
```

Или восемь (по стандарту ядра Linux):

```
public boolean hasStableIds() {
        return true;
}
```

Как правильно переносить код по достижении длины строки?

```
scroll.setLayoutParams(new LayoutParams(LayoutParams.FILL_PARENT,
LayoutParams.FILL_PARENT));
```

Или так:

```
scroll.setLayoutParams(new LayoutParams(LayoutParams.
FILL_PARENT, LayoutParams.FILL_PARENT));
```

В каких местах кода нужно добавлять пустую строку?

```
protected void onCreate(Bundle savedInstanceState) { super.on-
Create(savedInstanceState); setContentView(R.layout.thread_list);
Bundle b = getIntent().getExtras();
byte[] boardS = b.getByteArray("board");
ByteArrayInputStream bitch = new ByteArrayInputStream(boardS); Object-
InputStream in;
final View progWrapper = findViewById(R.id.threadlist_watcher_wrapper);
progress = (ProgressBar)findViewById(R.id.threadlist_watcher); progress.
setMax(100);
progress.setProgress(0);
new AlertDialog.Builder(KCThreadListActivity.this)
```

Или так:

```
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.thread_list);

Bundle b = getIntent().getExtras();
byte[] boardS = b.getByteArray("board");
ByteArrayInputStream bitch = new ByteArrayInputStream(boardS);

ObjectInputStream in;

final View progWrapper = findViewById(R.id.threadlist_watcher_wrapper);
progress = (ProgressBar)findViewById(R.id.threadlist_watcher);
progress.setMax(100);
progress.setProgress(0);

new AlertDialog.Builder(KCThreadListActivity.this)
```

Нужно ли разные условия писать на разных строках?

```
if ((null == fileName) || (null == content)) {
    return;
}
```

Или так:

```
if ((null == fileName) ||
    (null == content)) {
    return;
}
```

Как редактор должен обозначать конец строки?

Различия между операционными системами становятся все меньше. Еще пару лет назад перенос текстовых данных на другую операционную систему сопровождался искаженными чередованиями гласных, а сегодня Юникод расставил все на свои места. По крайней мере Mac OS X и Linux имеют сейчас одинаковые символы конца строки в текстовых данных (перевод строки) и разделительные символы для путей данных (косая черта, `/`). Так, в большинстве программных инструментов в Windows для обозначения путей данных можно использовать обычную косую черту вместо обратной.

Хорошие текстовые редакторы на всех платформах поддерживают действующие договоренности о конце строк. Только в проектах, предназначенных исключительно для Windows, вместо стандарта Unix используется символ line feed (перевод строки) ASCII 10, 0A (в шестнадцатеричной системе), а в исходном коде на многих языках он обозначается `\n`. На Windows строковый тип line feed часто сопровождается символом carriage return (LF CR), за ASCII 10 следует ASCII 13, в шестнадцатеричной системе за 0A — 0D, а в исходном коде за таким параметром следует `\n\r`.

Стоит проследить, чтобы ваш текстовый редактор придерживался стандарта расстановки конца строк, используемого командой проекта, иначе данные исходного кода, обработанные вами, будут отображаться у ваших коллег в виде одной очень длинной строки, что вызовет большое неудобство.

Насколько же важны различия в применяемых стандартах? Было проведено несколько исследований, в которых изучались различия в удобочитаемости кода с разным форматированием. Если сравнить такие тонкости с огромными различиями в удобочитаемости плохого и относительно неплохого кода, то окажется, что они имеют абсолютно одинаковое значение. На то, чтобы решить, какой длины должен быть отступ или что лучше использовать, пробелы или табуляцию, у вас уйдет чуть больше двух минут (если вы прочитали нашу книгу, конечно). Но очень важно твердо определиться с выбором одного из вариантов и не применять никакие иные. Не нужно придумывать авторский стиль — выбирайте один из закрепившихся. Он наверняка уже проверен временем, его использование будет оправданно, а программистов, читающих ваш код, «будет меньше от него тошнить»[1]. Обзор стилей форматирования можно найти здесь: de.wikipedia.org/wiki/Einrűckungsstil.

---

[1] *Goodliffe P.* Code Craft. — No Starch Press, 2006. — P. 26.

Договоренности в команде

Когда вы начинаете работать с новым языком программирования или над новым проектом, рекомендуется выполнять действия в два этапа.

1. Сначала смотреть, как это делают другие.
2. Затем делать все точно так же.

Следование правилам работы, принятым внутри фирмы или проекта, помимо всего прочего, целесообразно по той причине, что не нужно тратить много времени на форматирование кода других. Преимущества того, что все придерживаются одних правил, намного весомее, чем недовольство отдельных программистов, связанное с необходимостью сменить привычки. Хотя этим недовольством нельзя пренебрегать в случаях одновременной работы над несколькими проектами с разными принципами работы. В любом случае разум становится более гибким, если периодически менять свои привычки, связанные с форматированием кода, покупать кетчуп от разных производителей или посещать страны с левосторонним дорожным движением.

В сфере информационных технологий работает много людей, внимательных к различным деталям. Эта черта ценится в программировании, но может и повлечь за собой излишнюю вспыльчивость при работе с кодом, написанным другими программистами. Если у вас нет острой необходимости в изменении форматирования чужого кода, вы сделаете всему остальному миру одолжение, оставив все как есть.

Если чужой код нужно обязательно отформатировать, ни в коем случае не нужно делать это вручную. Если ваш текстовый редактор не поддерживает функцию автоисправления форматирования, вам нужна еще одна программа. Ее выбор зависит от используемого языка программирования, искать ее можно с помощью поисковых запросов `code beautifier` или `code formatter`. Некоторые текстовые редакторы и среды разработки имеют функцию меню Исправить форматирование или что-то подобное. В таком случае можно выделить участок кода и с помощью этой функции усовершенствовать его в соответствии с действующим на проекте стандартом. Это практично, но имеет значительное обратное действие, если у самой программы и людей, работающих над ней, разные представления о правильном форматировании.

В сфере программирования не принято указывать коллеге X на несоответствие действующим правилам форматирования, если вы используете функцию автоисправления. Даже системы контроля версий и инструменты diff (см. раздел «Diff и patch» главы 20) не всегда распознают то, что по-разному отформатированные данные имеют одно содержание. Поэтому в таких рабочих ситуациях следует применять автоисправление в самом начале, до того как код впервые проверяется системой контроля версий. В конце концов, нужно периодически настраивать правила автодополнения — это пригодится, если в одной команде не все исправляют форматирование одинаково.

Некоторые языки программирования могут смягчить такие недоразумения с помощью собственных технических возможностей. Так, в Python отступы не вызывают много споров, потому что в языке предусмотрено автоформатирование отступов. Для языка Go был разработан инструмент gofmt, в котором код тестируется перед запуском или проверкой в системе контроля версий. Этот инструмент

форматирует код в соответствии с едиными правилами и таким образом устраняет поводы для пустяковых стычек в команде.

Несмотря на существование технических возможностей замять конфликт, привычное вам форматирование кода все равно рано или поздно кто-то раскритикует. Лучше не приводите аргументы в пользу собственной точки зрения, потому что у вашего оппонента будут свои аргументы и вы будете тратить дни и недели впустую, вместо того чтобы написать хоть одну строчку кода. Обычно правила форматирования задаются автором первой строки кода. Если же возникают проблемы, вы можете проявить мудрость и уступить либо отказаться от совместной работы. Дорогие читатели, пожалуйста, не отстаивайте слепо свою правоту. Иначе мы, авторы книги, точно не достигнем своей цели.

5 Присвоение имен

> В информатике существует две реальные проблемы: управление кэшем и присвоение имен.
>
> *Фил Карлтон, программист Netscape*

Многие не очень хорошие программисты не слишком утруждают себя и не тратят свое драгоценное время на выдумывание имен переменных и функций. Они хотят побыстрее получить результат, и на этом пути им ничто не должно препятствовать. Поэтому они выбирают имя, которое хорошо подойдет для ситуации и будет кратким, например `tmp` или `var1`. В крайнем случае сойдет и `dings`.

Другие ужасаются, когда им нужно назвать переменную или функцию. По их ощущениям, должно произойти что-то значимое, что-то, что повлияет на весь ход развития проекта. Решения даются им тяжело, а выбранное имя может и не иметь никаких преимуществ. Для следующей функции сценарий повторяется. Через несколько дней оказывается, что первое имя не совсем уживается со вторым, все не работает. Но уже поздно что-либо менять, приравнивание имен, по всей вероятности, создаст только проблемы.

Тех, кого хоть однажды терзали сомнения, стоит ли переименование переменной потраченного на это времени, тех, кто задавался вопросом, стоит ли использовать краткие или содержательные имена, тех, кто встречал в чужом коде настоящих монстров типа `lpszStreetNameKey`, наверняка гложет мысль о том, что подробное рассмотрение этой темы пошло бы только на пользу.

Правила присвоения имен

Одна из проблем присвоения имен заключается в том, что для этого аспекта деятельности существует множество правил. Во многих языках программирования действуют собственные, жесткие и не очень, стандарты. Каждая фирма и даже каждая команда, занимающаяся проектом с открытым исходным кодом, имеет собственное представление о структуре имен.

Пока вы еще не очень хороший программист, вам стоит в первую очередь полагаться на правила языка, с которым работаете. Если выполнить поиск, например, с помощью запросов naming conventions или style guide и указанием предпочитаемого вами языка программирования, можно получить несколько документов, которые

послужат хорошим подспорьем. Даже крупные фреймворки, такие как Ruby on Rails и JEE или PHP-фреймворки Symphony и Zend, могут дать вдохновение для поиска.

Каждый язык имеет свой стиль написания имен. В связи с этим возникают два важных вопроса: «Что писать прописными буквами, а что — строчными?» и «В составных именах слова разделяются нижним подчеркиванием или с помощью CamelCase?». Имена констант, к примеру, во многих языках целиком пишутся прописными буквами. Таким образом, в этом вопросе есть совсем немного сходства и всеобщих предписаний.

Примеры имен в Python: `module_name`, `package_name`, `ClassName`, `method_name`, `ExceptionName`, `function_name`, `GLOBAL_VARIABLE_NAME`, `instance_variable_name`, `function_parameter_name`, `local_variable_name`.

Имена классов и исключений пишутся в стиле CamelCase, для глобальных переменных используются прописные буквы, остальные имена пишутся строчными буквами и разделяются нижним подчеркиванием.

Проблема, с которой до сих пор сталкиваются даже хорошие программисты, — это обращение со сложносоставными словами, то есть понятиями, состоящими из нескольких слов. В английском языке такие понятия пишутся раздельно. Если еще можно встретить двоякое написание `backgroundColor` или `background_color` из двух отдельных слов, то в случае `file name` это все не так уж и однозначно: способы написания `getFileName` и `getFilename` все еще борются за господство с вариантами раздельного написания `get_filename` и `get_file_name`. А если есть исключение для `filename`, как тогда написать `filePath`? И как назвать переменную, содержащую имя любимого животного, `favoriteAnimalname`?

Эта проблема связана с главным правилом, по которому имена функций и переменных должны соответствовать последовательной схеме, чтобы программисту можно было легко их угадать и тем самым сэкономить время. Нужно в целом ориентироваться на определенный принцип, всегда помнить о нем и не углубляться в тонкости названий. Мы рекомендуем быть последовательными и разделять каждое новое слово в названии большими буквами или нижними подчеркиваниями, то есть писать `getFileHandleForPathName` или `get_file_handle_for_path_name`.

На вопрос, в каком числе должны стоять имена переменных, нет общепринятого ответа. Если речь идет о присвоении имени коллекции, например массиву, содержащему цвета (red, green, blue, yellow), то более подходящим будет имя во множественном числе, поскольку эта коллекция содержит несколько объектов. С другой стороны, через некоторое время часто придется использовать элементы этой коллекции в единственном числе: `if (isPrimaryColor(colors[$i]))`... Одни программисты утверждают, что имя в единственном числе грамматически более правильно, другие же предпочитают узнавать непосредственно из названия, что переменная содержит несколько объектов. Попробуйте также решить, какой вариант вам ближе (или возьмите на вооружение принцип, действующий в команде), и оставайтесь ему верны.

Как писать имена функций и переменных и какая у них должна быть структура, несложно решить раз и навсегда, но при условии, что предварительно будут найдены правила для конкретного языка. Подбор подходящих имен — куда более трудное дело, которому программист посвящает всю свою жизнь.

Сначала Византий, потом Константинополь, теперь Стамбул

Размышления об именах требуют времени, но альтернативные пути отнимают его еще больше. Плохо подобранные имена приводят в будущем к значительной мыслительной активности и поиску ошибок в неправильных местах. Вот наиболее показательный случай, о котором рассказано на портале The Daily WTF:

> ...Через несколько строк нужно загрузить шаблон под именем InternationalRefTemplate. [Несчастный программист] потратил двадцать минут на знакомство с шаблоном InternationalRefTemplate, пока не понял, что речь-то идет о шаблоне InternationRefTemplate. Ни один из вариантов нельзя было путать с InternatinoalRefTemplate, в имени которого ранее была преднамеренно допущена ошибка, поскольку некто заблокировал данные об исходном коде InternationalRefTemplate на целые выходные, а другому разработчику пришлось вносить изменения.
>
> *thedailywtf.com/Articles/Poke-a-Dot.aspx*

Даже опытные программисты редко с ходу находят подходящее имя переменной или функции. Иногда с самого начала выбирают не самое лучшее имя, иногда по прошествии времени могут поменяться характеристики или требования кода. Возможно, в `elapsedTimeInMinutes` теперь отображаются секунды, поскольку нужно было более подробное уточнение. Преимущества того, что имена изменены, в любом случае преобладают над ущербом нашему эго, причиненным признанием собственной ошибки, поскольку вклад в пластичность кода, то есть облегчение процесса изменений кода, всегда того стоит и в долгосрочной перспективе значительно повышает качество кода.

Чтобы не потерять рассудок из-за постоянных переименований, в таких случаях важно иметь под рукой редактор, который будет активно помогать вам в этом деле. При использовании среды разработки с функциями рефакторинга (см. главу 20) жизнь становится намного легче. Когда есть возможность легко и без последствий переименовывать переменные, функции и классы в границах типов данных, груз ответственности при переименовании становится меньше. Если вы используете систему контроля версий (см. главу 21), то принятые вами решения не будут иметь губительных последствий, поскольку вы сможете их поменять, если на следующий день обнаружите, что они неэффективны.

В том случае, если программист использует простой редактор и усердно работает над изменением имен вручную, ему нужно застраховать себя от изменения хитрых имен, что может привести к каким-то сбоям. Так ему не придется много думать и искать, если он работает один, или объяснять, если работает в команде.

Предупреждаем тех читателей, которые в этом месте подумали: «Как хорошо, что мой редактор имеет функцию поиска и замены данных!» Поиск и замена данных — это игра с огнем. Прибегая к этому инструменту, вы не можете ничего изменить на протяжении остатка дня. «Да-да, я слежу за этим», — думаете вы. Это ваше право, но если вы однажды установите среду разработки с функциями рефакторинга, то поймете, почему мы об этом говорили.

В чем же различие между простыми поиском и заменой и функциями переименования в среде разработки (часто находятся в меню рефакторинга)? Последние понимают код и могут, к примеру, отличать имена переменных от имен функций. Интегрированная среда разработки также устроена довольно умно: замена `plane` на `jetPlane` не приведет к замене функции `findIntersectionOfLineWithPlane` на `findIntersectionOfLineWithjetPlane`.

Анализируйте значительные переименования, такие как при рефакторинге, и убедитесь, что уже написаны юнит-тесты для нужных участков кода (см. главу 16). С помощью тестов вы всегда сможете проверить, имеют ли неудачные изменения имен какие-либо последствия, влияющие на работу кода.

Работая в команде, нужно обсудить запланированные переименования с заинтересованными коллегами. Довольно часто у кого-то более богатая фантазия, чем у нас, кроме того, можно избежать всеобщей дезориентированности наутро после тяжелой ночи возни с именами.

Последствия присвоения плохого имени

В одном институте биотехнологии имелась таблица базы данных проведенных экспериментов. Одна колонка этой таблицы называлась is_ill, в ней указывалось, заразился ли подопытный организм болезнью. Первоначально были заданы следующие значения: NULL — неизвестно, ill — болен, healthy — здоров. Для функций или колонок таблицы с названиями is_x обычно используют значения типа Boolean, то есть true или false (см. раздел «Переменные типа Boolean» далее в этой главе. — *Примеч. авт.*).

Уже с самого начала выбранные имена приводили к путанице и, вероятнее всего, вызывали дальнейшие затруднения: при расширении схемы базы данных экспериментальные группы классифицировались как виртуальные эксперименты, у которых атрибут is_ill имел значение parent. Позже шаблоны экспериментов стали классифицироваться как эксперименты, у которых атрибут is_ill имел значение template.

Опытные сотрудники знали о таком развитии событий и особо не задавались вопросами. Однако новые сотрудники с волнением писали SQL-запросы, которые должны были перехватывать эксперименты типа template из базы данных. Нигде нельзя было найти колонки type, которая как раз предназначалась бы для этой цели. Это удивление было ничем по сравнению с тем, которое возникло у специалистов, когда они нашли типы экспериментов в графе is_ill. Было бы более разумно при расширении базы до экспериментов template и parent добавить еще одну колонку и атрибут experiment_type. Почему это не было учтено, никто не может сказать. Возможно, нужно было сделать все в кратчайшие сроки и предыдущий разработчик боялся, что расширение таблицы приведет к сбоям в программах, имеющих доступ к этой базе данных. Позже уже нельзя было бы переименовать колонку is_ill, поскольку для этого пришлось бы вручную переписывать ряд программ, использующих эту базу данных.

Реляционные БД не обладают производительными инструментами рефакторинга, поэтому изменения в схеме приводят к изменениям в приложениях, использующих базу данных. И поэтому изменения в схему вносятся еще реже, чем в программный код.

Йоханнес

Что должны уметь имена

Как подобрать хорошее, подходящее имя для переменной или функции? На практике ответ на этот вопрос нечасто услышишь и от хороших программистов. Никак. Намного быстрее, конечно, ввести в оборот `temp`[1], `tmp`, `x`, `xxx`, `xx1`, `variable`, `var`, `arr`, `value`, `val`, `thing`, `stuff`, `narf`, `bla`, `number`, `this`, `that`, `something`, `whatever`, `dummy`, `one`, `two`, `three`, `foo` или `bar`[2]. У многих есть свой список любимых имен переменных, используемых в случаях, когда ничего лучшего в голову не приходит. В кругу друзей авторов распространены следующие имена переменных: `сиропоткашля`, `салатизогурцов`, `скалка`, `куринаянога`, `кроликмилашка` и `размазня` («Это было раньше, теперь не так!»). Иногда с присвоением имен переменным нужно справиться быстро, потому что требуется сделать много других важных вещей, которые могут быть упущены, если долго думать только над именами. Можете писать и `сиропоткашля`, если вам это нужно сейчас же. Но при этом немного постыдитесь и после выполнения более важных задач все-таки посвятите время переименованию. Вам может помочь временное имя `ПеременнаяКоторойНеМогуПридуматьПодходящееИмяСейчасИПодумаюНадНимПотом` — оно такое длинное, что во время его набора вам может прийти в голову что-нибудь более уместное. Присваивать некачественные имена не смертельно. Смертельно их не исправлять в будущем.

Когда и после долгих размышлений вы не можете подобрать подходящее имя, это может свидетельствовать о том, что вы не совсем понимаете цель функции или переменной. Если для функции напрашивается имя `validateFormDataAndSubmitToServerOrDisplayErrorDialog()` и это имя правильно описывает ее, то это самое подходящее имя. Вдруг возникает ситуация, когда функция неверна. Не пытайтесь подобрать более красивое, простое, краткое или абстрактное имя без изменения самой функции. Наиболее оптимальное решение проблемы — выписать имя и пытаться изменять его длину до тех пор, пока не соберешься с мыслями, чтобы улучшить функцию. Такое улучшение проще пареной репы: функция разделяется на несколько функций — `validateFormData()`, `submitFormToServer()` и `displayErrorDialog()`.

Если лень набирать текст, вам может прийти в голову мысль назвать упрощенную функцию `processFormData`. В таком случае она будет сопровождаться комментарием длиннее, чем ее длинное имя, который включал бы в себя задачу функции и данные ввода/вывода. Аргумент «Да, но я могу написать его только один раз и все» не принимается, поскольку это означало бы, что программист, читающий код, при запуске функции не имеет никакой информации о ее предназначении.

---

[1] Почти в 0,0001 % случаев temp используется для того, чтобы присвоить имя переменной, интегрированной в код лишь на некоторое время. Это происходит, к примеру, когда нужно заменить значение двух переменных — для временного складирования данных используется третья.

[2] foo, bar и baz являются типичными метасинтаксическими переменными. Это значит, что они служат хранилищем для настоящих имен переменных подобно Lorem ipsum, который используется под графиками вместо реального текста.

Злые близнецы

В коде блога Riesenmaschine есть функции обработать_запрос и переработать_запрос. Вероятно, можно еще больше напортачить только в двух именах функций, но даже на примере этой конкретной ситуации можно сделать некоторые выводы.

Во-первых, имена следует писать по-английски. «Так я могу легче различать, какие функции написала я, а какие принадлежат PHP», — когда-то думала я. «Но ведь это и так видно в самом редакторе, он помечает их разными цветами». — «Ну да, конечно».

Во-вторых, в данном случае в одной всеобъемлющей функции объединено множество действий, и поэтому она имеет такое расплывчатое название: обработать_запрос. И в-третьих, через некоторое время понадобилась еще одна объемная функция, которую я бы назвала обработать_запрос. Поскольку это имя уже занято, вторым решением стало новое имя, похожее на первое. И конечно же, оба имени отображаются в URL-адресах Riesenmaschine, чтобы об этом знал весь мир.

Катрин

Читать, понимать и не путать

Если имя требует комментария, поясняющего, что первая буква в IllII — это прописная I, а не строчная L, что за ней следуют две строчных L и римская цифра II, то перед вами целое поле для совершенствования. Высока ли вероятность того, что имя поймут правильно, услышав его при обсуждении кода в переполненном баре или во время разговора по телефону в поезде? Избегайте, в частности, однобуквенных имен переменных. Единственными исключениями из правила являются временные переменные `integer` в циклах, такие как `i`, `j`, `k`, и геометрические координаты `x`, `y`, `z`. Аббревиатуры допускаются, только если они более широко известны, чем их полные наименования, например URL и HTML.

Есть и другие исключения из этого правила. В следующих случаях не принято использовать полную форму, так что пишите:

- `num` вместо `numberOf`;
- `pos` вместо `position`;
- `len` вместо `length`;
- `max` вместо `maximum`;
- `min` вместо `minimum`;
- `temp` или `tmp` вместо `temporary`;
- `val` вместо `value`.

Сокращать `count` до `cnt`, как это часто встречается, — плохая идея: хоть вы и набираете минимум текста, но перестаете понимать смысл. Также не пытайтесь заменить *ID for the Insurance Object Type* сокращением `idiot`[1]. В некоторых проектах

[1] Пример от Горки Сильверио: c2.com/cgi-bin/wiki?BadVariableNames.

существуют списки используемых сокращений. Часто попадаются сокращения, которые широко известны в какой-то сфере, но не обязательно — программисту. Если вы работаете в команде, заранее поинтересуйтесь, имеется ли подобный список у них.

Аргумент, часто приводимый не в пользу говорящих имен переменных и функций, — «это позволяет экономить время при наборе текста». Конечно, `RHC` напечатать быстрее, чем `RawHandlerContext`. И чем короче имя, тем меньше вероятность, что программист сделает в нем опечатку. Опечатки обычно быстро обнаруживаются, поскольку ошибочный код не будет работать, то есть это не очень грубые ошибки. Проблемы с непониманием кода, вызванные введением сокращений, приводят к куда более серьезным ошибкам, которые нельзя распознать автоматически, и на их обнаружение уходит больше времени, чем на набор длинных имен. Кроме того, найти при необходимости `k` или `rbs` гораздо тяжелее, чем `mostAdorableKittenName` или `numberOfRubysInStoneCollection`.

Рекомендуется найти золотую середину: хорошие редакторы кода могут предугадывать конец слова, если только введенное начало слова однозначно. Они выполняют поиск в открытых текстовых данных по подходящему слову и продолжают только что введенный ряд символов, если дать соответствующую команду сочетанием клавиш. Достаточно ввести, например, `RawH` и нажать клавишу Escape, чтобы написать `RawHandlerContext`. (В языках, где стандартом является CamelCase, можно просто ввести прописные буквы и затем запустить автозаполнение (табл. 5.1), `RHC` будет распознаваться как `RawHandlerContext`.)

Таблица 5.1. Вызов автозаполнения имен в различных редакторах

Редактор	**Сочетание клавиш**
Vi	Ctrl+P
Vim	Ctrl+N
Emacs	Esc, затем /
Eclipse	Alt+/
Intellij IDEA	Alt+/
TextMate	Esc
RubyMine	Ctrl+пробел
UltraEdit	Ctrl+пробел
XCode	Автозаполнение без нажатия клавиш
Sublime Text	Автозаполнение без нажатия клавиш, быстрое автозаполнение: Ctrl+пробел

Легко запоминаемые имена уже не так важны, если использовать среду разработки или один из перечисленных редакторов, поскольку в таком случае нужно лишь помнить, как имя начинается, а с остальным справится автозаполнение кода. Но тогда важна вот какая деталь: пытаясь искать `bonusPointsMax` по запросу maxBonusPoints, можно сделать своей памяти одолжение и дать искомому компоненту название, под которым привычнее осуществлять его поиск.

Если подобранные имена не схожи между собой, можно избежать путаницы, к тому же ошибки, допущенные по невнимательности, обнаруживаются быстрее. Отличительные черты имен не должны скрываться в дебрях длинных имен: `setFillColorForUpperPageMarginOfCoverArticle` и `setFillColorForLowerPageMarginOfCoverArticle` выглядят одинаково, если на них мельком взглянуть. Лучше было бы дать имена `setUpperPageMarginFillColorForCoverArticle` и `setLowerPageMarginFillColorForCoverArticle`, поскольку так Upper и Lower более заметны.

Имена ни в коем случае не должны различаться только регистром. И разница в одной-единственной букве слишком незначительна. Это актуально еще и тогда, когда эта буква, помимо всего прочего, является окончанием, обозначающим число. К примеру, `getUsername` и `getUsernames` очень похожи, `getAllUsernames` может решить эту проблему. Цифры в конце имен переменных свидетельствуют о том, что у автора не было желания хорошо подумать над ними. Цифра 1 выглядит во многих шрифтах почти так же, как строчная l, из-за этого обстоятельства один из авторов книги чуть не потерял рассудок.

Ясность и логика

Хорошие имена описывают не только форму, но и содержание. Недостаточно, например, две графы в структуре сайта назвать `left_column` и `right_column`. Однажды кому-то в голову придет идея расположить графу навигации на другой стороне. Таким образом, `left_column` и `right_column` создадут проблему, когда как с `navigation_column` и `text_column` такая проблема исключена. (По крайней мере до тех пор, пока кому-нибудь не придет идея располагать изображения в текстовой графе.)

Имена функций должны описывать, что эта функция выполняет, чего не выполняет, как она это выполняет. `highlightCurrentPage()` лучше, чем `setBackgroundOfCurrentPageToWhite()`. В последнем случае придется изменять все случаи запуска функции, если будет решено, что желтый — более заметный фоновый цвет или что только мерцание может привлечь нужное внимание.

Подумайте, в чем заключается суть переменной или функции, которую нужно назвать. Переменную, которая содержит имя файла для предварительной настройки программы, следует назвать `settingsFileName`, поскольку в этом заключается ее суть. `settingsText` не охватывает всю суть содержимого, ведь, если сами данные содержат текст, это нельзя назвать содержимым переменной. `programSettings` также не передает смысла, заложенного в переменной. Если возникают сомнения, выбирайте более специфические характеристики: можно даже в малых функциях с понятным кодом значению, полученному в результате арифметического вычисления, присвоить имя `result`, и это не вызовет серьезных последствий. Было бы лучше все-таки дать переменной говорящее имя, например `medianValue`.

Если информацией важно располагать для правильного обращения с переменной, очень важно, чтобы эти сведения были отражены в ее имени. Это особенно важно учитывать для единиц измерения: если вы назовете переменную `length_in_mm` или `delay_seconds`, вы передаете другому программисту, читающему код, важную информацию, что не смог бы сделать комментарий, поскольку единица из-

мерения видна на каждом участке кода, где используется переменная, комментарий же фигурирует только там, где она декларируется. На первый взгляд это вынуждает нас удлинить имя без особой на то нужды, но те, кто уже сталкивался с различными единицами измерения в веб-дизайне (px, em, pt), знает, сколько страданий могут принести недопонятые величины.

С помощью условий, указанных в имени переменной, можно выяснить и другие дополнительные сведения в тех случаях, когда велика вероятность неприятных сюрпризов. Назвать переменную `untrustedString` или `text_UTF8` может показаться слишком хлопотной и чересчур серьезной мерой предосторожности. Тем не менее это стоит делать, особенно если вы уже хоть раз совершали ошибку из-за нехватки этой информации. С одной стороны, вы вряд ли упустите из виду какую-то строчную переменную с суффиксом `UTF8`, если занимаетесь обработкой текста в кодировке UTF-8, с другой — не нужно бояться делать это, если вы, например, читаете и конвертируете строки в кодировке ISO-8859-1 и одновременно продолжаете использовать строки в кодировке UTF8.

Негласный закон, принятый во многих языках программирования, гласит, что имена констант пишутся прописными буквами. Другая закономерность, установленная в объектно-ориентированных языках, гласит, что имена переменных — членов класса `private`, то есть характеристики объекта, которые должны быть скрыты, начинаются с нижнего подчеркивания, например `_state`.

Хорошее имя должно отличаться ясностью. Не нужно использовать `dog`, если хотите в этой переменной сохранить *имя* определенной собаки. И наоборот, переменная `frameColor` должна содержать только цвета. Если нужно в этой переменной сохранить еще операции и образцы, лучше назвать ее `frameFill`. Но не следует повторять имена классов в переменных-членах и геттере/сеттере. Если вы занимаетесь объектно-ориентированным программированием и пишете класс `Animal`, окрас не обязательно обозначать как `animalColor`, достаточно будет просто `color`.

Если функции взаимосвязаны, это должно быть понятно по возможности с первого взгляда. Такие взаимосвязи можно продемонстрировать более показательно с помощью реструктурирования кода, но присвоение имен может помочь и здесь. Если одна функция должна первой выполнить определенные задачи, после чего процесс может продолжаться, эта функция в своем имени должна нести признак `initialize` или `prepare`: `initializeTimer`, `prepareConnection`.

Следите за взаимосвязанными переменными одновременно. Если одна переменная носит имя `color`, а потом оказывается, что нужно обозначение и для фонового цвета, появляется искушение назвать вторую переменную `backgroundColor` и покончить с этим раз и навсегда. Это непоследовательно и вызывает больше путаницы. Лучше переименовать и первую переменную, чтобы в итоге получилось `textColor` и `backgroundColor`.

Помните о возможном непонимании. Увидев переменную `map`, разработчики не станут разбираться, находятся там данные карт или это *ассоциативный массив*. Вероятность путаницы в данном случае возникает из-за того, что такие массивы есть в некоторых языках программирования, например в C++ и Java, и они называются `map`. `getIP` — не лучшее имя для функции с задачей get interesting people (пример взят из истории развития социальной сети aka-aki.com).

АССОЦИАТИВНЫЕ МАССИВЫ

Ассоциативный массив — это метод упорядоченного хранения ключевых слов и значений, например "name" => "Иван Петров".

Образность может особенно легко привести к путанице и недопониманию. В языках программирования, как и в естественных языках, существует множество метафор: *trees*, *frames*, *windows*, *Cloud*, классы `decorator` и `factory` в объектно-ориентированном программировании и многое другое. Новые метафоры стоит вводить только в случае крайней необходимости. Даже когда вам предельно ясно, почему понятия `dogsitter` или `navelgazer` лучше других описывают ваш класс, они могут добавить проблем программистам, читающим код, в особенности если эти программисты — представители другой культуры. Кроме того, возникает угроза выбора начинающими метафор, о которых они, в принципе, могут и не знать. В таком случае они выбирают метафору, которая для других программистов имеет особое значение. В качестве примера можно назвать `visitor`, `facade`, `observer` — общепринятые *шаблоны проектирования*, запрещенные к использованию в качестве имен переменных или классов. Полезно будет заблаговременно прочесть список шаблонов проектирования хотя бы для того, чтобы хоть раз просмотреть их имена. Статья в «Википедии» по адресу en.wikipedia.org/wiki/Software_design_pattern содержит их краткий обзор.

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ

Шаблон проектирования — это путь решения повторяющихся проблем. С шаблоном проектирования можно сравнить блокированную застройку в архитектуре. Это решает проблему, и при высокой плотности застройки дома получают какой-никакой сад. Если активно не применять шаблоны проектирования в разработке ПО, это может привести к пренебрежению расплывчатыми понятиями и именами, которые можно было бы поискать или обсудить с другими программистами.

Никаких шуток! Никакой крутости!

В отличие от других сфер деятельности человека, в программировании отсутствие чувства юмора и недостаток оригинальности только способствуют присвоению удачных имен. Да, иногда очень хочется назвать переменные `$c3po`, `$r2d2`, `Laserpony` и `0verlord`. Нет, шутки не будут поводом для гордости при следующей сверке кода, лишь возникнут вопросы о том, что может скрываться за той или иной переменной. Хорошо, если вы знаете, что профессионалы часто используют имя `i18n` для обозначения понятия internationalization, чтобы сэкономить место. Но все равно нужно записывать это слово полностью, поскольку не все программисты будут знать об этом. Убедитесь в том, что своими секретными символами вы не обидите заказчика, других программистов, использующих код, или изобретателя языка программирования. Наконец, все равно, что об этом думает любой незнакомый вам программист, но если ваш начальник или клиент намеренно или случайно просмотрит код, это может стать причиной их недовольства.

I18N

Число 18 появилось в слове internationalization, потому что между i и n находится 18 букв.

Незабываемый момент

Фирма моего друга была партнером Microsoft. То есть фактически Microsoft ее полностью финансировал. Там работали над одной игрой, но у них были большие проблемы с DirectX, который в то время только появился. Для одной крупной игровой ярмарки они подготовили демоверсию, не хватало только установщика. Его они написали ночью перед ярмаркой (сейчас должна играть музыка, предвещающая беду) и затем уехали туда уже с установочным CD.

Начинается ярмарка, и игра моего друга — это центральный элемент стенда Microsoft. Запущенная игра должна демонстрироваться на огромном пятиметровом экране над самим стендом.

Они пытаются установить игру на компьютерах с ОС Microsoft, но… она не устанавливается.

Владелец фирмы уже на месте. Куратор проекта от Microsoft — тоже. В выставочном зале толпы людей. Технические специалисты Microsoft пытаются помочь с устранением проблемы. Может, виноваты драйверы? Или версия ОС? Никаких изменений. Опять никаких. Они открывают Registry с помощью regedit. Да, проблема таится здесь. Переменной BILL_GATES_IS_AN_******* присвоено неправильное значение в установщике.

И все это на пятиметровом экране в regedit. В Microsoft шутку не оценили.

Анонимный комментатор,
c2.com/cgi/wiki?BadVariableNames

Материал, из которого делаются имена

Не очень хорошие программисты и люди, для которых английский язык не родной, придумывают новые слова для тех понятий, которые уже получили определенное название от опытных программистов. О шимпанзе Вашоу, освоившей около 350 слов американского жестового языка, сообщается, что она для непонятных ей явлений сама придумала названия, такие как *water bird* или *metal cup drink coffee*. Это превосходно для шимпанзе. Но человеку стоит понимать, что для этих понятий есть уже укоренившиеся обозначения, а именно *«лебедь»* и *«термокружка»*. При поиске ответов на вопросы в Сети (см. главу 8) тоже полезно знать специальный термин для того, что нужно найти.

Поэтому в следующем разделе мы приводим несколько широко используемых элементов имен, которые не так просто узнать с ходу или различия между которыми непросто определить самому. Это их неполное собрание, но после прочтения этого раздела у вас хотя бы будет представление о том, что происходит в голове у хороших программистов в процессе присвоения имен. Возможно, вы

обратите особое внимание на те элементы имен, которые встречаются вам в чужом коде, и включите их в собственный активный словарный запас для программирования.

Процессы, функции, методы

Функции обычно обозначают какое-то действие. Это должно быть отражено в их именах, поэтому начинайте их с глагола. `markDirty()` смотрится лучше, чем `dirty()`.

DIRTY

Dirty в данном случае обозначает, что какой-то пакет данных был изменен и его нужно, например, сохранить. Чтобы не заблокировать программу, вполне разумно отложить процедуру сохранения. Если же пакет данных обозначен как dirty, позже становится известно, что именно хотели сохранить. Это своего рода метафора, но программисты прибегают к ней.

Объектно-ориентированное программирование

Большое исключение из этого правила действует для объектно-ориентированного программирования. Объекты составляют здесь некое пространство имен (см. раздел «Пространства имен» главы 26), в которое входят все методы. Поэтому во многих объектно-ориентированных языках это имя имеет вид Объект.сделать!, что обозначает: в пространстве имен (контексте) объекта нужно действовать следующим образом. Всегда нужно писать Abbreviation.expand(), если речь идет об объекте Abbreviation, но ни в коем случае не Abbreviation.expand_abbreviation(), поскольку методы объекта должны иметь непосредственное отношение к объекту — еще одно упоминание Abbreviation было бы лишним.

Мы составили список глаголов, которые часто употребляются в именах функций. Некоторые из этих глаголов могут быть защищены от использования в зависимости от конкретного языка программирования, например, extract в PHP. Это значит, что так нельзя назвать собственную функцию. Список помогает найти глагольную часть имени, но решает только часть проблемы, ведь за глаголом обычно следует существительное: `getQuestion`. Если это не приведет к еще большей путанице, можно также вставить и прилагательное, как, например, в `getUnansweredQuestions`, и/или после существительного ввести еще одно определяющее слово, как в `getUnansweredQuestionsList`, `showUnansweredQuestionsTable`.

Поиск и извлечение

- `get`.

  Примеры: `getParent()`, `getName()`, `getRemainingFuel()`.

  Значение просчитывается, когда это необходимо, и функция выводит его на экран. В объектно-ориентированных языках общепринято для инкапсуляции

переменных объекта писать целый ряд методов типа getter, которые предоставляют доступ для чтения этих в противном случае недоступных переменных. Эти методы обычно называют по формуле `getVariablename()`. В некоторых языках, например в Objective-C, принято убирать префикс. Если же этого не требуют ни правила языка, ни ваши коллеги, лучше прописывать `get`, поскольку это более конкретно.

Если функция начинается с `get`, опытные программисты могут из этого заключить, что имеющийся в виду процесс будет выполняться относительно быстро и без затруднений. В данном случае это не так, поскольку функция, к примеру, просматривает в течение минуты сеть или весь жесткий диск. Лучше выбрать глагол, который будет обозначать эти временные затраты, например: `compute`, `acquire`, `fetch`, `download` или `retrieve`. Так программисты, активизирующие функцию, будут предупреждены об этом.

Противопоставление по признаку синхронности и асинхронности также играет некую роль при выборе подходящего глагола. Если включить в скрипт URL и в какой-то момент получать в выводе данные от этого URL, браузер будет некоторое время находиться в состоянии бездействия. Здесь речь идет об асинхронной операции. Глагол `request` часто служит показателем того, что в данной функции заложена асинхронная операция. `get` никогда не обозначает асинхронную операцию, `request` — очень часто, `query` может обозначать оба вида операций.

- `find`, `search`.

  Примеры: `string.find` в Python, `array_search` в PHP.

  `find` осуществляет поиск внутри коллекции (данные в каталоге, элемент в списке, иголка в стоге сена). Результатом обычно служит вывод местоположения искомого объекта в коллекции или особого значения, например `null`, `nil`, `0`, `-1`, в случае, если объект не найден. Некоторые программисты могут попытаться использовать `seek` вместо `find`. `seek` используется для других целей, она откладывает операции чтения или записи, выполняемые читающей/пишущей головкой на другом участке потока данных.

  Другой глагол, `search`, также может вызвать проблемы понимания: search a bag означает «поискать внутри сумки», а не «искать сумку». При использовании `search` в именах функций нельзя добавлять в него искомый объект, а нужно определить место, где должен производиться поиск: `searchFolder` ищет *не сам* каталог, а *внутри него*.

- `include`.

  Пример: Server Side Includes (Web).

  Значение, особенно часто встречающееся при компиляции в таких языках, как C, или более старой серверной веб-технологии Server Side Includes: данные должны быть встроены на место команды `Include` до их дальнейшей обработки.

  `include` может быть частью имени функции, если вы хотите этим показать, что читаются и встраиваются другие источники данных, но чаще `include` является

частью имени переменной типа boolean, такой как `includeOldData`, которая показывает, включать или не включать определенные источники данных в обработку.

- `scan`.

  Пример: `scandir` в PHP.

  От функции, имеющей в имени `scan`, стоит ожидать, что она не будет выводить первые попавшиеся ей под руку подходящие данные, а будет просматривать таблицы или жесткие диски полностью и только затем предоставлять вывод данных. Даже если еще не известно точно, что нужно искать, `scan` может стать хорошим решением проблемы. В то время как другой поисковый глагол, например `find` или `search`, означает, что в целом мы просто быстро просматриваем какой-то индекс, то `scan` говорит: осуществляется поиск чего-то важного и это может продолжаться долго.

Вывод

- `dump`.

  Примеры: `var_dump` в PHP, `dumpAccountData`.

  Собранный объект станет по крайней мере более понятен при использовании `dump`, но будет не совсем красиво выводиться на экран, обычно только для отладки, но никак не для конечного пользователя.

- `render`.

  Примеры: `renderFrame`, где под `Frame` имеется в виду отдельное изображение, `renderHTML` (в противовес `parseHTML`).

  Некое преобразование данных, предварительная обработка, которая облегчает их дальнейшее использование (например, людьми). Примеры: объект базы данных форматируется в виде HTML-кода, программа ray tracing выделяет некое изображение из данных 3D. `render` иногда отличается большой длительностью (вам может хватить времени, чтобы принести и выпить кофе).

- `show` и `hide`.

  Примеры: `showTooltip`, `hideTooltip`.

  Операция `show` применяется в случаях, противоположных тем, в которых применяется `hide`. Если же нет вариантов с `hide`, то лучше использовать `display`.

Объединение

- `join`, `merge`.

  Пример: `array_merge()` в PHP.

  С помощью `join` или `merge` несколько частей данных объединяются в единое целое. `merge` послужит более оптимальным вариантом, поскольку для `join` существует много правил, продиктованных контекстом: в рамках баз данных это означает, что результаты из двух таблиц выводятся вместе на основании некоего сходства. Массивы могут объединяться также с помощью `join`, но для большинства языков это означает, что они, к примеру, выводятся в одной строке,

при этом обособляясь запятыми. `Node_is` включает в себя функцию `Path.join()`, которая умело объединяет несколько участков потоков данных в новом потоке данных и удаляет при этом не имеющие смысла записи; `os.path.join` в Python работает аналогичным образом. `merge` подразумевает, что вывод, полученный от операции, будет иметь тот же тип, что и данные ввода.

- `append`, `concatenate`.

  Примеры: `append` в Python, `concat` в JavaScript.

  Прикрепляет один объект к концу другого, например одну строку к концу другой или один файл к концу другого. Если прикрепить новую часть данных в начале, а не в конце, то в ход идет `prepend`. Функция `concatenate`, часто сокращаемая до `concat`, имеет похожее действие. Использование `append` обычно обусловлено работой с *коллекцией*, к которой необходимо прикрепить некий объект. Так, например, эту функцию принято использовать в jQuery. `concatenate` в целом прикрепляет одну коллекцию к другой и таким образом служит особой разновидностью `combine`. При выборе функции будет полезно ответить на вопрос, должна ли функция работать как некая математическая операция и выводить данные, не изменяя при этом сами операции: расчет 2 + 3 = 5 оставляет 2 и 3 без изменений. `Append` изменяет первую часть операции: посредством объединения `stringA` и `stringB` мы получаем удлиненную строку `stringA`. `concatenate` *может* изменить `stringA`, но не стоит этого делать. Часто из-за этого получается абсолютно новая `stringC`.

КОЛЛЕКЦИИ

Коллекция — это обобщающее понятие для структур данных, в которые входят элементы одних и тех же классов данных, например массивы, списки, карты, словари, хеш-таблицы.

- `combine`.

  Примеры: `combine_arrays`, `combine_tables`.

  Очень размытое понятие, которому иногда действительно нужно быть таким размытым. `combine` объединяет два и более объекта в один и часто используется, если списки, массивы или другие собрания данных должны быть сведены в нечто единое. Самим понятием не выдвигаются требования к тому, как именно это сведение должно происходить. Например, можно написать функцию, которая объединяет два массива в один. Если в массиве A есть элемент, который встречается и в массиве B, убрать эту копию было бы вполне разумно в зависимости от области использования этих массивов. Такие действия могут быть заданы с помощью следующего параметра:

```
function combine_arrays (var a, var b, var strip_duplicates) {
  ...
}
```

Более узкоспециализированными разновидностями, которые могут неким образом описывать характеристики организации комбинируемых данных, являются `splice` и `concatenate`.

Для простых массивов, созданных вами самостоятельно, редко целесообразно писать функцию такого рода, поскольку все широко используемые языки программирования обладают функциями слияния массивов и сортировки дубликатов. Будучи ученым, вы можете, к примеру, объединить две базы данных генома различных организмов и гены, присутствующие в обеих базах данных, объединить в одну запись, поскольку это может послужить ценной дополнительной информацией для ваших пользователей. В этом случае можно было бы назвать функцию не `strip_duplicates`, а, скажем, `combine_matching_genes`.

Разделение

- `split`.

  Примеры: `split` в Perl и Python.

  Делит объекты на несколько частей, обычно это разделение значений одной строки с помощью запятых (или других разделительных знаков). Операция, противоположная по действию `join`.

- `slice`, `splice`

  Примеры: `slice` в Python, `splice` в JavaScript, PHP и Perl.

  Обе операции позволяют вычленить одну часть коллекции, например, из массива. В выводе всегда отображается лишь фрагмент. При этом важно учитывать, не пропал ли теперь этот фрагмент в исходной коллекции — такой вариант операции называется `splice`. В ходе работы эта операция может заполнять пробелы новыми элементами, как это происходит в процессе *плетения* канатов. Использование `slice` или `splice` в именах функций помогает другим программистам, читающим код, всегда придерживаться выбранного имени. В случае сомнений обычно прибегают к помощи библиотеки стандарта используемого языка программирования.

Преобразование

- `parse`.

  Примеры: `parse_url` в PHP, `JSON.parse` в JavaScript.

  По сути, эти операции служат для парсинга грамматики. В JavaScript `JSON.parse` из строки в формате JSON создает новый объект. В PHP `parse_url` логично разделяет URL на компоненты и выводит массив с элементами, такими как `[path]`, `[query]` или `[port]`. HTML-парсеры преобразуют строку формата HTML в *DOM-узел*. Операции, абсолютно противоположной `parse`, не существует; в зависимости от особенностей преобразованные данные можно описать с помощью `render`, `serialize` или `toSomething` (см. далее).

DOM (DOCUMENT OBJECT MODEL)

DOM — это интерфейс, обеспечивающий пользователю доступ к структурированным данным в документах форматов HTML и XML. Используя его, мы работаем не с текстовой кашей, а со структурой дерева. Все дерево представляет собой узел документа, элемент HTML типа <p> представлен в виде узла элемента, а его содержимое — в виде узла текста.

- `serialize`.

  Примеры: `serialize` и `unserialize` в PHP.

  Сериализация создает из объектов одну структуру, которую, к примеру, можно вписать в текстовый файл. Помимо `serialize`, этот процесс может называться также `stringify` (например, `JSON.stringify` в JavaScript). При присвоении имен собственным функциям `serialize` является наиболее широко используемым понятием. Противоположная функция называется `deserialize` или `unserialize`. В Python эта пара функций называется `pickle` («помещать») и `unpickle`, в Perl — `freeze` и `thaw`.

- `toSomething`.

  Примеры: `toHTML`, `toJPG`, `toString`.

  Широко используемое сокращение для `convertToSomething`.

- `invert`, `reverse`.

  Примеры: `reverseSortDirection`, `invertColors`.

  `reverse` изменяет очередность действий на противоположную. `invert` преобразует положительные значения в отрицательные, светлые серые тона — в темные, синий цвет — в желтый.

Начало и прекращение работы

- `start-stop`, `begin-end`, `open-close`.

  Что касается этих функций, здесь стоит полагаться на правила парного комбинирования. Не следует использовать `beginTransaction` в паре с `closeTransaction` или `openFile` в паре с `destroyFile`. В противном случае более опытные программисты будут читать вашу работу как предложение, которое началось за здравие, а закончилось за упокой.

- `pause`, `stop`, `exit`.

  Примеры: `pauseMovie`, `stopListening`, `exitState`.

  Самыми однозначными именами в данном случае являются `stop` и `exit`. `pause` используется тогда, когда есть возможность использовать `resume`. Также в случае работы с процессами, связанными с нашим восприятием, мы можем использовать пару `pause-play`.

- `destruct`, `destroy`, `kill`.

  Примеры: `destruct` и `session_destroy` в PHP, `kill` в командной строке.

  Функции удаления отличаются степенью своей «жестокости». `destruct` служит антонимом функции `construct` и имеет собственные правила: сначала разрушается крыша, потом фасад, а затем все это убирается вниз в определенном порядке. Функция `destroy` также выполняется в определенной последовательности. Во главе этих функций стоит `kill`, которая применяется только в исключительных обстоятельствах, в том случае если другие функции никак не могут помочь.

Очистка и перемещение

- `flush`.

  Пример: `flushCache` переписывает данные кэша на диск.

  Некие данные где-то (обычно в *буфере вывода*) на некоторое время были сохранены, а сейчас их из этого временного хранилища извлекают и обрабатывают, то есть, к примеру, они записываются на жесткий диск или выводятся на дисплей. Это служит частью меры оптимизации: есть некая дорогостоящая операция, например запись на жесткий диск, которую нужно отложить до лучших времен, когда она действительно себя оправдает. Недостатком при таком положении дел является невозможность сохранить данные: если перед выполнением `flush` произойдет отключение электричества, все данные будут потеряны.

- `clear`, `purge`.

  Примеры: `purgeTempTables`, `clearFilters`.

  Происходит очистка некоего хранилища, скажем буфера или кэша. В отличие от `flush`, `clear` не подразумевает дальнейшей обработки содержимого этого хранилища. `flush` отправляет старое стекло из дома на переработку, помещая его в соответствующий контейнер для отходов, а `clear` просто-напросто оставит это стекло на лестничной площадке — главное, что место освободилось. `purge` действует еще решительнее, чем `clear`, и после нее остаются следы нежеланного содержимого.

- `cleanup`.

  Пример: `cleanupTempFiles`.

  Незавершенные процессы или процессы в режиме ожидания принудительно завершаются — свободные нити завязываются в узлы.

- `collapse`, `compact`.

  Примеры: `collapseNavigation`, `compactDatabase`.

  То, что может занять много места, объединяется. Часто это связано с вполне объяснимыми обстоятельствами, например в случае с `collapseNavigation`, однако иногда без более абстрактного объяснения не обойтись: `collapseEmptyTags` меняет `<foo></foo>` на `</foo>`. Если мы встречаемся с `collapse`, то часто можно встретиться и с соответствующей ей функцией, начинающейся с параметра `expand` и отображающей совмещенные данные в выводе. Глагол `compact`, в свою очередь, означает, что объединение данных, по всей вероятности, необратимо (если, например, в некоем списке удаляются все элементы со значением `NULL`).

- `move`.

  Пример: `move_uploaded_file` в PHP.

  Основное внимание здесь редко уделяется движениям, видимым невооруженному глазу (за исключением использования функции в играх). Как правило, `move` описывает процесс перемещения, например, когда данные из одного источника перемещаются в другой и сохраняются в нем. В старом месте хранения их найти нельзя, иначе это уже `copy`.

- `tidy`, `pretty`, `beautify`.

  Примеры: `tidyHTML`, `prettyPrint`, `beautifyJSON`.

  Здесь подразумевается приведение данных в порядок в плане их визуализации, а не в плане управления ресурсами. Данные обрабатываются для дальнейшего нормального восприятия человеком, например, в текст добавляется больше пробелов, отступов и интервалов, более того, часто они вставляются автоматически. Если нужно применить противоположную функцию в целях экономии места и времени, необходимого для отображения данных, используется `minify` или `uglify`.

Превратить временное в долговечное

- `save`, `store`, `write`.

  Примеры: `saveToTempFile`, `storeResult`, `writeToSocket`.

  `save` и `store` закрепляют данные в переменной или долговечном хранилище (долговечно оно по крайней мере до тех пор, пока на него случайно не прольется чай из кружки или его не коснутся несколько мощных магнитов). `write` также подразумевает, что данные записываются в канал передачи данных без цели их где-либо сохранить.

- `commit`.

  Примеры: `commitAllTransactions`, `commitNewItems`.

  `commit` применяется тогда, когда производятся некие транзакции, то есть выполняется сбор произведенных изменений и они записываются вместе. Поскольку этот глагол используется в SQL в качестве команды, он плотно взаимодействует с банками данных.

- `bake`.

  Пример: `bakeDetectedFrames`.

  Наличие `bake` в имени функции говорит программисту, читающему код, о том, что процесс может длиться больше, чем планировалось, и при этом к коду прибавляются объекты, которые впоследствии нельзя будет изменить. Например, в редакторе видеофайлов можно наложить на изображение субтитры отдельным слоем, что длится довольно долго. `bake` «впаивает» их на века. Таким образом, в данном случае гибкостью пренебрегают ради быстроты процесса.

Нет, подождите! Верните все как было!

- `undo`.

  Пример: `undoMove`.

  Наиболее очевидный вариант имени для функций, отменяющих некие понятные действия. За `undo` должен следовать глагол, но ни в коем случае не названия структур данных (например, `undoData`, `undoTree`), что часто встречается в некачественном коде. У многих программистов любимым способом образования новых глаголов, обозначающих функции, служит добавление приставки `un-` к уже используемому глаголу: `unDelete`. Это часто применяется к глаголам, уже

означающим обратимость некоего действия: `unRevert`. Функцией, противоположной `undo` по действию, в любом случае является не `unUndo`, а `redo`.

- `rollback`.

  Пример: `rollback_patch_installation`.

  Как правило, служит для обозначения обширной операции, которая имеет значение в первую очередь для транзакций в базах данных. `rollback` всегда предполагает полную обратимость предыдущих действий с транзакцией (см. раздел «Транзакции и откаты» в главе 26). Используйте эту функцию, только если вы можете гарантировать, что процесс полностью отменит предыдущую операцию. Любой откат может повлиять на несколько объектов. `undoTransaction` не так хорошо подчеркивает сложность происходящего, как `rollback`.

- `restore`, `revert`.

  Примеры: `restoreOriginalSettings`, `revertAllChanges`.

  В данном случае системе возвращается ранее присущее ей состояние или состояние по умолчанию. Значит, это состояние нужно было предварительно где-то сохранить. `revert` имеет определенное сходство, однако в то время, как `restore` имеет непосредственное отношение к состояниям и объектам, `revert` связана с внесенными изменениями: в переводе с английского *to restore a state (a configuration, a file)* обозначает «восстановить состояние (конфигурацию, файл)», но *to revert changes* — «восстановить изменения». `revert` обычно применяется в комплексе с системой управления версий, в случае сомнений лучше использовать `restore`.

- `recover`.

  Пример: `recover_session`.

  Обычно применяется в порядке исключения в тех случаях, когда совершаются попытки что-то воспроизвести заново, например восстановить случайно удаленные данные. Вывод в таком случае угадать нельзя. `recover` обычно выступает в роли красного из M&M's (см. главу 14). Те, кто попытался внедрить эту функцию в собственный код, в следующий раз подумают дважды, прежде чем еще раз это сделать, и в случае сомнений откажутся от этой идеи. Да лучше уж умереть на месте, чем где-то исказить данные.

С этими именами лучше не шутить

- `process`, `do`, `perform`, `dealWith`, `manage`, `change`, `handle`, `resolve`.

  «Вы сможете применить processData только единожды на протяжении всей своей карьеры, потому что после этого вас, строго говоря, уволят», — пишет К. Л. Уэнхэм, автор статьи на sites.google.com/site/yacoset/Home/naming-tips. Не все придерживаются такого мнения, например, методам часто присваивают абстрактные имена по довольно веским причинам. Возьмем библиотеку класса аудиофильтра: все аудиофильтры имеют определенное сходство — они изменяют аудиоданные. Данные входят в один канал и выходят из другого уже в измененном виде. В таком случае ничто не мешает нам назвать метод `process` в одном базовом классе, определяющем схожие черты этих фильтров. Такие сознательно нетщательно отобранные

глаголы обычно встречаются в библиотеках и реже — в коде приложений. Если вы не знаете, что вам делать в таком случае, и не можете принять приемлемое решение, лучше избегайте многозначных имен. Такие глаголы заставляют программиста набить функцию множеством различных задач, а пользователь кода не может никак узнать, какие именно задачи выполняются.

- `filter`.

  Функции фильтрации данных всегда обладают неким критерием фильтрации, однако, к сожалению, есть две договоренности о том, что этот критерий должен показывать, будет ли кофейный остаток, то есть данные, не прошедшие фильтрацию, использоваться дальше. Тогда лучше будет применить `select`, `find` или `extract`. Или функция выбросит этот остаток и просто «приготовит кофе»? Тогда подойдут `reject`, `exclude` или `strip`.

- `check`, `validate`, `verify`, `test`.

  «Проверь, не убежало ли молоко», — говорит один другому. «Я проверил, оно убежало ровно в половину двенадцатого», — отвечает второй. Если `checkError` имеет в результате `true`, то это означает, что мы так и не стали умнее. Из этих глаголов совершенно непонятно, что должна выводить функция и в какой форме должен отображаться вывод. Часто в подобных случаях можно добиться большей однозначности с помощью переменных типа Boolean, таких как `isValid`, `isDatabaseConnectionOpen`.

Переменные типа Boolean

Для функций с результатом, выраженным типом данных Boolean, то есть возвращающих `true` или `false`, используется префикс: `isValidZipCode()`. Многие программисты пренебрегают этим правилом и дают таким переменным имена типа `undo`, `empty`, `gender`, `direction`, `disable` или `status`. Не стоит следовать этому примеру. С одной стороны, префикс `is` дает невнимательным программистам, читающим код, следующую информацию: здесь стоит ожидать результат `false` или `true`. С другой стороны, так можно научиться более естественно выражаться в собственном коде. `if (isEnabled && isValid)` читается лучше, чем все остальное. Некоторые языки, такие как Ruby, как правило, в именах функций содержат `?: mayAccess?`.

Глядя на имена таких переменных, нужно без долгих раздумий уметь распознавать, что будет означать результат. Так, результат можно хорошо проиллюстрировать в функциях `isDone`, `hasError`, `isValid`. Труднее становится тогда, когда в имени переменной содержится отрицание: `isUnsuccessful`, `isDisabled`. Это легко приводит к путанице с двойными отрицаниями в условиях («Данные не были не найдены?»). Не избегать отрицания в именах переменных нерационально.

В редких случаях может оказаться разумным поместить в начало имени `was`, `has`, `can` или `should` вместо `is`: `wasMaximizedAtExit`, `hasPermission`, `canEdit`, `shouldAbort`. Хорошие программисты в основном догадываются о назначении функций, поскольку `is` однозначно и легко узнаваемо. Кроме того, `is` наводит программиста на размышления о самой переменной, а не о ее окружении, поэтому `isMutable` выглядит лучше, чем `canBeModified`. `can` вызывает еще одну проблему: подразумевает ли автор под `canUndo`, что это действие `isUndoable`, или он тем самым просто дает пользова-

телю право на это действие? Каждая секунда, которую читатель кода тратит, чтобы разобраться в этом, — ценное время. Не стоит, однако, писать `isAbleTo` только для того, чтобы не использовать `can`.

Один лишь факт того, что процесс завершился несколько секунд назад, еще не служит поводом для использования `was` (`wasFound`), поскольку, строго говоря, мы лишь узнаем состояние переменной в прошлом. `was` подходит для использования в тех ситуациях, например, когда некое предыдущее состояние должно быть восстановлено для использования. Часто в конце концов мы все равно пытаемся найти приемлемый вариант с `is`.

Присвоение имен может сказаться на защите данных

Curl-API содержит параметр CURLOPT_SSL_VERIFYHOST. Этот параметр определяет, нужно ли проверять SSL-сертификаты.

CURL

Curl обозначает Client for URLs (клиент для URL) и является широко используемым инструментом для переноса данных в компьютерных сетях. При написании приложений для сети Curl обычно применяется для того, чтобы извлечь данные из некоего источника, а затем обработать их в собственном приложении. API (интерфейс программирования приложения) — это интерфейс программирования, в этом случае для библиотеки libcurl.

SSL-СЕРТИФИКАТЫ

SSL-сертификат играет некую роль, если вместо HTTP при разработке веб-приложений используется HTTPS и вместе с ним зашифрованное соединение. Это происходит, например, при разработке приложений онлайн-банкинга.

Программисты предполагают, что здесь речь идет о переменной типа Boolean, поскольку в конце концов в этой ситуации нужно решить, проверять или не проверять сертификаты. Если VERIFYHOST = 0, возникает следующая вполне ожидаемая ситуация: Curl не смотрит на сертификаты. Если же переменной присваивалось значение 1 или true, то вплоть до конца 2012 года происходило следующее: Curl проверял, встречались ли в сертификате какие-либо имена хостов, однако не сравнивал эти имена с актуальным именем хоста, а принимал сертификат в любом случае, и неважно было, кто его предъявлял. Он вел себя так же, как служащий, который проверяет удостоверения сотрудников, но пропускает их, даже не вглядываясь в фотографию.

Такие действия Curl привели к полному пересмотру вопросов безопасности, связанной с работой SSL.

Подобающая проверка SSL-сертификатов стала происходить только после того, как VERIFYHOST было присвоено значение 2. Это кажется необычным, однако проблемы вообще не возникло бы, если бы разработчики назвали переменную, к примеру, HOST_VERIFICATION_LEVEL. Такое имя сразу же сообщает каждому программисту, что здесь могут существовать две возможности и лучше проконсультироваться по этому вопросу с документацией.

Имена переменных во множественном числе вызывают у многих программистов определенный дискомфорт: `areShelvesEmpty`, `areHamstersHungry`. Нужно их применять только в том случае, если все остальные варианты плохо согласованы грамматически или широко не используются. В объектно-ориентированном программировании выбор имени во множественном числе свидетельствует о том, что можно еще больше разделить класс, поскольку, к примеру, он представляет собой массив или перечень отдельных объектов. На вершине ответвления дерева класса не должны встречаться имена во множественном числе, там должны фигурировать классы `Shelf` или `Hamster` с собственными методами `isEmpty` или `isHungry`. Коллекция объектов, которые ранее были собраны в одном классе, должна быть пересобрана в массив или перечень, который будет вмещать в себя объекты `Shelf` и `Hamster`.

Очень редко не подходит ни один из используемых префиксов Boolean. `if (userExists)` в одном обсуждении пользователей портала Stack Overflow[1] было определено как наименее плохое решение, все же лучшее, чем `isUserExist`, `doesUserExist` или `isUserExisting`. Но не стоит при всяком удобном случае прибегать к этим именам, а лучше делать это, если вы сможете при необходимости обосновать свой выбор.

Полезным глаголом для переменных Boolean служит `toggle`. Он преобразует состояние в противоположное и в целом работает, как хорошо знакомый нам электрический выключатель. Два запуска `toggle` восстанавливают исходное состояние. Примеры: `toggleVisibility`, `toggleExpertMode`.

Объектно-ориентированное программирование

В объектно-ориентированном программировании имена классов выражаются существительными. Эти существительные часто совпадают по звучанию с профессиями (`Manager`, `Writer`) или устройствами (`Converter`, `Adapter`). Таким образом, это определяет то, что в объектно-ориентированном программировании объекты рассматриваются в качестве неких предметов, которым свойственно некое поведение. В имени не нужно указывать на то, что речь идет о классах, поэтому лучше не писать по формуле `НекийClass`. Если в классе применяется шаблон проектирования, его в имени класса все же стоит указать: `ConnectionFactory`. Как и в именах функций, существительному может предшествовать прилагательное и, возможно, какое-то описание структуры. С не самыми показательными, но полезными примерами такой структуры имен можно ознакомиться на сайте classnamer.com. В поиске имен также поможет просмотр какого-нибудь фреймворка, например *Spring*.

В некоторой степени выбор существительных обосновывается окружением, в котором находится код. Если вы пишете код для приложения сервиса доставки еды, вам, скорее всего, понадобятся классы `Slicer` и `Butterer`, которые, кроме

---

[1] stackoverflow.com/q/1566745/2535335.

вас, никто не использует. Однако чаще, судя по всему, стоит применять технические концепты, для которых уже где-то были установлены определенные понятия.

Часто в объектно-ориентированном программировании интерфейсы получают имена в форме прилагательных, образованных от глаголов, например `Comparable`, `Runnable` или `Serializable`. Это означает, что применительно ко всем объектам, использующим определенный интерфейс, можно совершать некие действия. Объекты, применяющие `Comparable`, характеризуются тем, что их можно сравнивать друг с другом, объекты `Serialize` можно сериализовать для того, чтобы, например, их можно было легче записать на жесткий диск.

`Helper`, `Manager`, `Util`, `Processor`, `Handler`, `Service`, `Coordinator`. Об этих определениях классов много спорят. Их противники считают их слишком расплывчатыми и общими, говорящими о том, что некто подменяет свое небогатое процедурное мышление мнимым ориентированием на объект. Кроме того, за такими именами часто скрываются классы, которые не отвечают принципу единичной ответственности, то есть слишком много на себя берут. Их сторонники утверждают, что эти компоненты имени широко распространены (фреймворк .Net содержит, к примеру, классы `Manager` в больших количествах), поэтому по ним хоть как-то можно узнать, что же это за класс. А значит, не нужно их использовать так самонадеянно — стоит рассмотреть несколько других вариантов.

Базы данных

Стандарты для SQL- и реляционных баз данных были учреждены еще до появления WWW. Это, к сожалению, привело к тому, что вокруг каждого известного продукта создавалась собственная культура со своими обычаями. Между этими культурами не было большого обмена, поскольку специалист по базам данных Oracle никогда не работал бы в фирме, занимающейся dBASE или MS-SQL. В настоящее время все еще действуют SQL-стандарты, а в Сети уже происходит некий обмен, однако различные традиции укоренились. В некоторой степени это происходит по причинам, связанным с технической составляющей вопроса. Так, в Oracle команды SQL и имена таблиц нечувствительны к регистру, а преобразуются внутри системы в прописные буквы, но только если стоят в кавычках. Postgres нечувствителен к регистру, но преобразует все ключевые слова в названия, написанные строчными буквами, а mySQL, в зависимости от операционной системы, движка баз данных и настроек, может быть как чувствительным, так и нечувствительным к регистру. Поэтому по всему миру много времени отводится на раздумья о том, как найти адрес клиента: `customer_address`, `CustomerAddress`, `customerAddress`, `tbl_customeraddress` или `T_CUSTOMER_ADDRESS`. Если вы начинаете работать в существующем уже некоторое время проекте, безотказно следуйте принятым в нем правилам, даже если считаете, что эти смертные вас недостойны.

Многие общепринятые стандарты использования баз данных в настоящий момент требуют, чтобы имена таблиц стояли во множественном числе. Таким образом, таблица с именами людей должна быть названа Persons. Часто имена таблиц и колонок принято писать в так называемом *PascalCase* или *UpperCamelCase*, то есть

начинать слова с прописной буквы и разделять их прописной буквой: `FamousPersons`. Также широко используется традиция писать слова в именах полностью прописными или строчными буквами, разделяя их нижним подчеркиванием, например: `FAMOUS_PERSONS`. В Rails предусмотрен следующий образец написания имен таблиц: `invoice_items`. Внимательно следите за правильным употреблением множественного числа: таблица с именами людей может называться как `Persons`, так и `People`. Возможно, эта таблица содержит графу `PersonName` или из другой таблицы на нее ссылаются с внешним ключом `PersonId`, тогда `People` — не самый удачный выбор.

Имена таблиц и столбцов не содержат пробелов. Имена столбцов употребляются в единственном числе, таким образом, в таблице `Users` есть столбец `UserName` и запрос будет выглядеть следующим образом: `SELECT UserName FROM Users WHERE UserID = 123;`.

Первичный ключ таблицы должен содержать ID: если таблица называется `Hedgehogs`, то первичный ключ — `HedgehogID` или `hedgehog_id`. Часто хочется (и это широко распространено) назвать первичный ключ просто `ID`, но, даже будучи не самым профессиональным программистом, можно вспомнить о случаях, когда приходится использовать данные из различных таблиц, столбцы которых имеют более говорящие имена. Большинство программистов ставят `ID` в конец имени. Те, кто предпочтет поставить его в начало, будет искать нужную информацию день и ночь и скорее застрелится, чем найдет, но в таком случае это правило должно соблюдаться всегда, и уже нельзя метаться между двумя вариантами.

Если таблица содержит внешние ключи из других таблиц, то есть то, что в другой таблице находится в столбце `id` и является там первичным ключом, этот столбец стоит называть в соответствии с таблицей, из которой берутся необходимые данные. Если используется таблица `Countries`, то следует ссылаться на ее ID в столбце с именем `country_id` или `CountryID`, то есть `SELECT * FROM Users JOIN Countries ON Users.CountryId = Countries.CountryId;`.

Если идентичное или схожее содержание появляется в нескольких местах в одной базе данных, то оно должно иметь одинаковое и, следовательно, предсказуемое имя. Если в одном месте почтовый индекс назвать `Индекс`, в другом — `ПочтовыйИндекс`, а в третьем — `ZipCode`, нужно постоянно проверять, какой вариант требуется в данном случае.

Для присвоения имени *соединительной таблице* имена обеих таблиц совмещаются в имени совместно используемой таблицы в алфавитном порядке: таблица `Courses` содержит названия предметов, `Students` — имена студентов, в таблице `Courses_Students` содержится информация о том, кто из студентов посещает тот или иной курс.

СОЕДИНИТЕЛЬНЫЕ ТАБЛИЦЫ

В соединительных таблицах отражается взаимосвязь типа «многие ко многим», например, между студентами и курсами: каждый студент может посещать несколько курсов, а на каждом курсе может быть много слушателей.

Таблицам поиска присваиваются имена таблиц, на которые они ссылаются, например, `User_roles` содержит значения `Administrator`, `Enduser`, `Developer` и кодирует тип пользователя, который может запросить определенные права. Так можно

легко догадаться, какие таблицы взаимосвязаны между собой, и нет никаких несоответствий имен между схожими поисковыми процедурами для различных таблиц.

ТАБЛИЦЫ ПОИСКА

Таблицы поиска, или словари, — это таблицы, преобразующие читабельные имена для данных ввода-вывода в числовые имена для осуществления операций баз данных.

Некоторые программисты баз данных изо всех сил стараются в каждое название столбца включать сокращенное имя таблицы: `cust_first_name`, `cust_last_name`. У этого есть свои плюсы: все столбцы имеют однозначные имена, и при рефакторинге очень легко осуществить поиск по всему коду на предмет наличия обращений к определенному столбцу. Недостаток заключается в том, что нужно набирать больше текста и запоминать, какое сокращение используется. В случае работы с внешними ключами это может привести к появлению чудовищных запросов наподобие `SELECT * FROM Orders JOIN Customers ON Orders.Ord_CustomerId = Customers.CustomerId WHERE Customers.Cust_LastName = 'Lauert';`.

Может оказаться полезно включать в имена столбцов с датами параметр `_date`, с временем — `_time`, а с временными метками — `_ts`. Не используйте никаких общих описательных слов для дат (типа `date`), старайтесь их сузить: `created_date` обозначает дату, для которой был создан отдельный фрагмент данных (возможно, в другой системе), `inserted_ts` — время вставки этого фрагмента в базу данных и т. д. В Rails, если столбцы содержат дату и время, их принято называть `created_at` и `updated_at`. Если в них стоит только дата, используются имена `created_on` и `updated_on`.

Некоторые разработчики используют в именах таблиц префиксы `T_` или `tbl_`, в именах представлений — `V_`, материализованных представлений — `MV_`. Мы все же не советуем этого делать, поскольку, если из соображений производительности нужно обычное представление преобразовать в материализованное или изменяются какие-либо компоненты базы данных и таблица заменяется представлением, то либо придется очень много исправлять, либо используемый префикс и тип данных просто не будут соответствовать друг другу. Чем использовать неправильные префиксы для описания типа, лучше не использовать никаких.

Что же дальше?

Если и после чтения этой главы вам все еще сложно присваивать имена, не стоит беспокоиться по этому поводу, напротив, это хороший знак. Биоинформатик Роланд Краузе, основываясь на своем опыте преподавания в университете, говорит: «Умные студенты говорят: мол, с именами у меня проблемы. Недобросовестные студенты вообще не понимают, зачем этой теме посвящать время». Если же вы поняли, что над этой темой действительно время от времени стоит задумываться, это говорит о том, что вы уже сделали большой шаг вперед.

Но не нужно при этом впадать в ступор. Если в течение десяти минут вам в голову не приходит ни одного хорошего имени, напишите какое-нибудь плохое, воз-

можно, в следующий раз у вас получится придумать что-то получше. Иногда проблему присвоения имен можно вовсе обойти, не задавая значениям собственные переменные, а создавая их каждый раз заново при работе с кодом. При возникновении сомнений с помощью PHP-команды `date('Y')` можно прочитать в документации, что в данном случае имеется в виду, если же вы сами придумали имя `this_year` или `now`, так не получится. Если даже вы, вводя имя `now`, сможете написать качественный комментарий к нему, он все равно не будет появляться при каждом упоминании переменной. Это ведет к определенным потерям производительности, если при каждом запуске одной строки что-то считается, измеряется или проверяется заново, но, как у любого новичка, у вас есть много других забот, помимо этой.

Не нужно слепо полагаться на то, что некоторые перечисленные здесь рекомендации будут соблюдаться в любом коде. Если автор неизвестного вам кода что-то сделал по-другому, не нужно сразу с высоты полученных вами знаний кричать, что вся его работа — полная чушь. Может быть, это и так, но не стоит исключать возможности того, что такие решения были приняты опытным программистом вполне осознанно.

6 Комментарии

> На самом деле при возникновении потребности в написании комментария к коду, видимо, появляется и понимание того, что же происходит.
>
> *Кристиан Хеллер/@plomlompom, Twitter, 3 декабря 2009 года*

Если вы сами программируете, очень важно, чтобы вы думали о том, что читательская аудитория вашего исходного кода лишь частично представлена машинами. Куда более важными читателями являются другие программисты. Они являются вашей непосредственной целевой группой. В отличие от компилятора вашего языка программирования, этой целевой группе комментарии зачастую намного более кстати.

Бьерн Страуструп, разработчик языка программирования C++, в своей книге «Язык программирования C++» утверждает: «Написать хорошие комментарии может быть таким же сложным делом, как и написать саму программу». Поэтому не стоит слишком строго себя критиковать, если ваши собственные комментарии, написанные две недели назад, кажутся загадочной археологической находкой, не имеющей ничего общего с современностью. Но проблемой пренебрегать нельзя, поскольку написание хороших комментариев — дело не только такое же сложное, как и написание хорошего кода, но и такое же важное.

На следующем примере рассмотрим, как из непонятного кода и, вероятно, путаного комментария можно прийти к удачному комментарию.

Этот код на языке C вполне понятен:

```
void finalize(RHC* rhc) {
    int i;
    for(i=0; rhc._mIdG->c > i; i=i+1) {
        free(rhc._mIdG->p[i]);
    }
}
```

Этот код сопровождается комментариями, которые занимают столько же места, но делают его ненамного понятнее для других:

```
/* --------------------
* void finalize(RHC* rhc)
* Финализация структуры RHC.
* rhc: ссылка на структуру RHC для финализации
```

```
* --------------------*/
void finalize(RHC* rhc) {
    int i;
    // перебор всех IdGs
    // и очистка их ссылок
    for(i=0; rhc._mIdG->c > i; i=i+1) {
        free(rhc._mIdG->p[i]);
    }
}
```

К сожалению, приведенный пример встречается во многих проектах. Комментарии, которые являются просто дословными переводами фраз кода на английский или немецкий, не только бесполезны, но еще и не повышают производительность. Их сокращение становится целой проблемой, когда программист переписывает код и при этом забывает соответствующим образом изменить комментарии. Занимаясь ночь напролет отладкой неработающего веб-приложения, можно легко изменить код, но при этом забыть о комментариях. Если приложение начинает работать, то уже никто и не вспомнит о том, что английский язык и язык C где-то расходятся в формулировках, а через год кто-то будет долго рвать на себе волосы и сомневаться в собственном здравом уме. При чтении собственных и чужих комментариев можно подумать, что они не имеют никакого отношения к коду. Лучше их рассматривать как тон пьяного человека: заинтересованный, но скептичный.

Код без комментария в первом случае непонятен не потому, что языки программирования в целом непонятны людям и требуют перевода в форме комментариев. Он непонятен, поскольку автор не сочувствует читателю кода. Он непонятен, поскольку автор использует несметное количество сокращений, для прочтения которых требуется детальное знакомство со всем кодом. Он непонятен, поскольку при чтении так и не стало понятно, что же за идея такая стоит за этим кодом, зачем он вообще нужен, — такой код не предоставляет контекста. В естественных языках такое количество причин непонимания обнаруживается только при чтении инструкции Deutsche Telekom о подключении NTBA к сплиттеру ADSL 2+ и терминалу данных.

Убрав сокращения и присвоив имя функции, которая характеризует задачу кода (см. главу 5), мы получаем такой результат:

```
void freeAllGlobalIDs(RawHandlerContext* rhc) {
    for(int i=0; i < rhc.globalID->count; ++i) {
        free(rhc.globalID->buffer[i]);
    }
}
```

Вообще мы понятия не имеем, что такое `RawHandlerContext`. Но по крайней мере уже знаем, что у него есть глобальные идентификаторы чего-то и что эта функция стирает эти идентификаторы. Это дает уже немало информации, причем совсем без комментариев.

Еще одна версия кода:

```
/* К сожалению, этот код здесь просто необходим, поскольку кажется, что
в RawHandlerManager есть ошибка, которая без этого кода приводила бы
к утечке GlobalId-буферов. Я не могу понять, почему это происходит.
```

```
Надеюсь, мы скоро починим этот баг, чтобы избавиться от этого временного
решения проблемы. 2008-07-23 -- regular@muskelfisch.com */
void freeAllGlobalIDs(RawHandlerContext* rhc) {
    for(int i=0; i < rhc.globalID->count; ++i) {
        free(rhc.globalID->buffer[i]);
    }
}
```

Вот оно как! Такой комментарий уже не повторяет ни слова из кода. Он рисует нам полную картину событий. Он содержит информацию, абсолютно ненужную машине и предназначенную исключительно для людей по той причине, что ее никак нельзя зафиксировать на языке программирования. Теперь мы знаем, где нам продолжать копать и даже кого мы можем об этом расспросить. И, что наиболее важно, знаем, что автору самому не нравится эта функция. Пара предложений дает нам ясное представление, которое мы, возможно, не получили бы даже из самого совершенного кода.

Меньше слов — больше дела

Относительно количества комментариев не определено каких-то общепринятых рамок. Никудышные программисты иногда вообще не комментируют код, однако это свойственно и многим хорошим программистам. Различие между двумя этими категориями людей заключается в том, что последние позже все-таки разберут, что они там написали.

> Я пишу много комментариев; некоторые утверждают, что чересчур много. Но мне очень нравится, когда код может рассказать мне обо всем, даже если мне четыре года. Мне не очень нравятся сплошные строки кода без интервалов. Если я вижу, что три-четыре строки связаны между собой, я после них оставляю интервал и краткий комментарий, дающий пояснения к следующему блоку. [...] Единственная проблема, которая может, на мой взгляд, возникнуть, — это свойство программистов забывать изменить комментарий при внесении изменений в код. Тогда мы имеем дело с ложным комментарием, а это еще хуже, чем его отсутствие, поскольку такой комментарий при чтении кода дает неверные ориентиры.
>
> *Ричард Мазородзе, разработчик ПО*

Эксперты в сфере программирования советуют не комментировать очевидное. Но все не может быть вполне очевидным для программистов с разным опытом. Новичкам сложно оценить, какой комментарий позже окажется полезным другим. Несмотря на широко распространенные рекомендации, мы советуем нашим читателям первое время комментировать как можно больше. Код не самого высокого качества с подробным комментарием куда лучше кода не самого высокого качества без комментария. Возможно, вам поможет тщательное выстраивание всех процессов в голове, будто вы маленький ребенок, который учится говорить и все проговаривает про себя: «Сейчас я кладу синий камешек на зеленый, а дальше...» Даже

через некоторое время, когда вы станете опытным программистом, вы не сможете угадать, кому придется читать ваш код. Может, это будет беспомощный студент, которому этот код нужно будет прочитать в рамках практики, и он так порадуется вашим комментариям очевидного. Если это происходит в единичных случаях, то нужно помнить, что неуместные комментарии стереть легче, чем неуместный код, и если вы это сделаете, то в другой части кода ничего не нарушится.

Если же вы видите, что у кода есть лишь несколько общих комментариев, незначительно проясняющих его, будьте внимательны. В связи с этим часто приводят слова уважаемых в сфере программирования Брайана Кернигана и Роба Пике: «Не фиксируйте ошибки плохого кода — сразу переписывайте его». Конечно, легче сказать, чем сделать, и такая ситуация может привести к тому, что, кивая на слова Кернигана и Пике, код не только не комментируют, но и не переписывают. Поэтому мы хотим немного продолжить вторую часть этого высказывания: не переписываешь код, так хотя бы прокомментируй его. Иначе в конце концов забывается, что в нем было плохого и как это получилось. Но вообще из-за таких неурядиц можно так разозлиться, что уж лучше переписать.

О структуре комментариев

Во многих языках есть два вида комментариев: однострочные, которые располагаются в конце строки кода, и многострочные, стоящие перед кодом и занимающие несколько строк.

Однострочные комментарии отделяются от кода специальными символами, в зависимости от языка программирования, например `//`, `#` или `--`.

```
// устанавливаем цвет по умолчанию
color = "#ff0000" // ярко-красный
if ( (rowIndex % 2) == 0) { // только четные ряды
    color = "#303060" // синеватый оттенок серого
}
```

Для многострочных комментариев во многих языках действует стандарт C++:

```
/* Для того чтобы наш стол радовал глаз,
 * мы используем два чередующихся цвета для ряда заднего вида
 * стола.
 */
```

Статья английской «Википедии» *Comment (computer programming)*[1] содержит обзор стилей написания комментариев в различных языках.

Начинающие часто используют только одну из разновидностей: либо многострочный, либо однострочный комментарий. В действительности же у обоих вариантов есть свои специфические сферы применения. Большинство комментариев проще найти, когда они находятся на отдельных строках. Во-первых, тогда их легче просмотреть при чтении кода, а во-вторых, появляется больше места

[1] en.wikipedia.org/wiki/Comment_(computer_programming)#Styles.

и тогда не нужно слишком активно сокращать комментарий в ущерб его смыслу. Однострочный вариант подходит для документации параметров и переменных и часто используется, к примеру, для указания единицы измерения и диапазона значений переменной:

```
const float speedLimit = 300.0; // метров в секунду
global int timeLastMoved = 0; // в миллисекундах
                              // с момента запуска программы
```

...много строк другого кода...

```
void moveSpaceship(
    int currentTime, // в миллисекундах с момента запуска программы
                     // (должен быть >=0)
    float deltaX, // горизонтальное движение в метрах
    float deltaY // вертикальное движение в метрах
) {
    spaceShip.x += deltaX;
    spaceShip.y += deltaY;
    // рассчитать расстояние, на которое удалилась ракета,
    // используя формулу a^2 + b^2 = c^2
    float distanceMoved = squareRoot(deltaX* detaX + deltaY* detlaY);
    // рассчитать количество секунд, прошедших с момента последнего
    // движения
    float deltaT = (currentTime – timeLastMoved) / 1000.0;
    // рассчитать скорость в м/с     float speed = distanceMoved / deltaT;
    // если нет записи о времени последнего вызова,
    // мы не сможем определить скорость ракеты
    if (timeLastMoved != 0) {
        // скорость ракеты выше допустимой?
        boolean tooFast = speed > speedLimit;
        if (tooFast) {
            ...
        }
    }
    // помнить о текущем времени для следующего вызова
    timeLastMoved = currentTime;
}
```

Здесь есть одна проблема: часто программист, читая код, не видит ни объявления переменных, ни сопровождающего его комментария, не говоря уже о том, что он его еще и не знает наизусть. Поскольку же, например, единица измерения `speedLimit` из приведенного примера упоминается только в объявлении, он должен посмотреть ее там, чтобы при сравнении с формой `speed > speedLimit` суметь отделить мух от котлет. Поэтому лучше упомянуть единицу измерения в имени переменной (см. главу 5). Тогда нужно лишь один раз глянуть и проверить правильность кода: `speedInMetersPerSecond > speedLimitInMetersPerSecond`.

Очень помогает ввод действительного диапазона значений, например, для параметра функции (в примере выше `currentTime` должно быть положительным зна-

чением). А еще лучше протестировать ввод на компьютере, для этого есть инструкция по Assert (см. раздел «Утверждения» в главе 26).

Многострочные комментарии должны стоять перед кодом. Это нужно для того, чтобы при рассмотрении критического блока кода можно было сначала подумать о комментарии и больше не ломать над ним голову. Например:

```
webSettings.setJavaScriptCanOpenWindowsAutomatically(false);
webView.setWebViewClient(new KCWebViewClient());
/* Обходной прием для
 * https://code.google.com/p/android/issues/detail?id=12987 "WebView
 * не работает на Android 2.3". Обходной прием взят с  * http://quitenotewort-
hy.blogspot.com/2010/12/handling-android-23-webviews-broken.html
 */
if ((Build.VERSION.SDK_INT == 9) || (Build.VERSION.SDK_INT == 10)) {
    javascriptInterfaceBroken = true;
    webView.setWebChromeClient(new KCWebChromeClient());
}
```

Комментарий относится к `if`-блоку и призван с помощью ссылок пояснить, почему здесь запрашиваются необычные условия для того, чтобы в определенных версиях программы отключить внедренные особенности.

Многострочные комментарии не нужно отделять межстрочным интервалом от строки, к которой они относятся (в данном случае к `if`-инструкции), для того чтобы можно было и визуально определить связь комментария с кодом.

Комментарии должны стоять максимально близко к строкам, которые они описывают, иначе высока вероятность, в том числе и практическая, того, что можно изменить их вместе с кодом. Поэтому лучше вместо целых романов в начале каждой функции писать небольшие комментарии.

Комментарии к документации

Описанные ранее примеры комментариев предназначены для программистов, читающих исходный код, и поэтому обычно они располагаются поблизости от того участка кода, который они описывают. Другой вид комментариев — так называемые комментарии к документации оформляются так, чтобы их можно было извлечь из исходных данных с помощью генераторов документации и преобразовать во внешний документ, например, в форме страниц HTML.

Комментарии к документации всегда стоят над переменными или функциями или в начале определений класса и описывают класс или непосредственно следующую функцию или переменную. Для компилятора это нормальные комментарии, которые с помощью дополнительных знаков форматирования подсказывают генератору документации, что они должны быть преобразованы как раз с помощью этих самых генераторов.

В Java наиболее эффективно применяются комментарии к документации: уже в первых версиях языка документация велась именно с помощью такой формы комментариев, в данном случае названной JavaDoc. Блок комментария JavaDoc, стоящий перед методом, выглядит примерно следующим образом:

```
/**
*  Конструктор и основная точка входа для чтения страницы платформы
*
*  @date: 10.11.200917:27:11
*  @author Johannes Jander
*
*  @param url URL для загрузки и парсинга
*  @param boardDbId уникальный ID платформы в базе данных
*  @return содержимое парсированной страницы
*/
public String KCPageParser(String url, long boardDbId) {
    ...
}
```

Обычно многострочные комментарии завершаются в Java с помощью `/* */`. Дополнительная звездочка в начале комментария говорит о том, что это комментарий типа JavaDoc.

Комментарии к документации обладают определенной структурой. Первый абзац содержит краткое описание класса или документируемой функции. Он должен дать программисту возможность использовать данный код без необходимости его прочтения, то есть он доносит информацию о предназначении кода. За функцией следуют ее параметры, обычно обозначаемые `@param`, и результат функции, обозначаемый `@return`. Другие поля, такие как `@author` или `@version`, используются не так часто.

В других языках существуют в чем-то схожие, в чем-то различающиеся генераторы документации: RDoc для Ruby, Docstrings для Python, Doxygen для C++. В Perl система документации (pod) работает по-другому, а в C# система под названием XMLDOC записывает комментарии к документации в XML.

Когда и что нужно комментировать

Комментарии были придуманы не просто для красоты, и они не делаются в самом конце работы, когда все остальное готово[1]. Люди со среднестатистической (читай: низкой) степенью самодисциплины и силы воли вообще опускают этот этап работы даже в самом конце. И даже если в конце концов решиться на такой шаг, половина мыслей, возникавших во время написания кода, к тому времени уже забывается. Поэтому лучше писать комментарий параллельно с кодом. Если приходится так сильно сосредотачиваться на написании кода, что уже не хватает внимания на комментарии, лучше вообще такой код не писать — наверняка он чересчур сложный. Некоторые авторы даже советуют писать комментарий до написания кода. Просто если в псевдокоде зафиксировать все, что планируется сделать, не получится запутаться в подробностях работы, а, напротив, появится время подумать о написании всего кода в общем. Псевдокод, таким образом, выступает в качестве пояснения для самого автора кода и других людей, которые его читают. Кроме того, при сопостав-

[1] Исключение: заяц, нарисованный ASCII-символами в конце файла.

лении комментария и кода можно проверять, действительно ли было написано то, что планировалось написать.

В качестве основного правила можно принять следующий принцип: комментарий должен содержать все, что программист хотел бы сказать своим коллегам при совместном разборе собственного кода. В идеале комментарий должен не дублировать все, что делает код, а описывать, что нужно делать коду и почему. Не будет ошибкой вместо особых путей решения проблемы описать общий концепт. (Преимущество данного подхода в том, что комментарий не нужно менять, если позже та же цель достигается другим, более элегантным путем.) Хорошо будет, если ваши комментарии содержат слово «поскольку». Не следует писать лишь «цикл назад по массиву», а следует — «цикл назад по массиву, поскольку сортировка по умолчанию при выводе замедляется». То, что массив запускается с самого конца, можно легко понять из самого кода, но почему это происходит, узнать нельзя.

Если вы перед собственными функциями и методами пишете комментарии, раскрывающие их смысл, то можете использовать следующую форму повествования: «Я хочу, чтобы эта функция, опираясь на заданное исходное значение, выводила 50 значений из таблицы USER. Если этих 50 значений нет, пусть она выводит все остальные. Если исходное значение больше количества значений таблицы, она не должна иметь вывода, а если ниже нуля, функция должна выводить ошибку». Эта не слишком сухая форма хороша тем, что вы не тратите много сил на описание того или иного кода. Вместо этого она помогает вам выразить свои требования и пожелания.

Наш пример взят из User Stories о разработке ПО Agile, в которых повествование всегда ведется от первого лица единственного числа (форма «я»). Часто используемая альтернатива — форма множественного числа («мы»):

```
// мы должны проверить, существует ли файл, до того как записывать в него
// что-то
```

Такое обращение может обозначать различных субъектов: «мы, разработчики кода», «я и техника» или «я, автор, и ты, читатель». К часто используемым структурам относятся обращение во втором лице:

```
// вам нужно изменить это значение на 0 для отладки
```

конструкции в страдательном залоге:

```
// значение должно быть изменено на 0 для отладки
```

и обобщенные формы:

```
// проверить, существует ли файл, и записать
```

Некоторые программисты склоняются к использованию страдательного залога, даже если пишут на английском. В этом есть определенный недостаток, поскольку часто мы не знаем, кто должен действовать в данный момент. Нужно всегда четко формулировать, кто является субъектом действия в данный момент: написанный код, запускаемый код, определенный объект, пользователь или таинственные силы вселенной.

Рассматривая этот вопрос, стоит сказать, что любое решение будет делом вкуса и здесь не так уж важно придерживаться единственного варианта. В некоторых

ситуациях так и просится какая-то определенная форма, например «вы», если читателям кода объясняют, как пользоваться API. Если вы хотите признать свою вину в позорном «хаке», лучше использовать первое лицо. Всегда при написании комментариев ставьте себя на место читателя, чтобы у вас не получилось такой ситуации как у mrgoat, который вот как пишет о такой форме комментариев: «“Не делайте это, не посмотрев вот то предварительно”, — раньше я и сам такие комментарии писал. Позже мне нужно было что-то исправить в коде, и я был просто обижен надменными примечаниями одного идиота-программиста. “Хм, — подумал я, — и ты мне что-то хочешь посоветовать, написав код из рук вон плохо!” А затем я понял, что код-то написан мной самим»[1].

Случаи, когда комментарий мог бы помочь

Если код совершает неожиданные действия

Обычно следует избегать таких ситуаций, но бывает, что вы используете чужой код, фреймворки или API, на которые почти не можете повлиять. Если что-то при первом прочтении кода не соответствует ожиданиям, во второй раз будет точно так же, поэтому такой участок нужно комментировать. Рассмотрим это на следующем примере (в Java)[2]:

```
File tempFile = new File(tempFStr);
FileOutputStream out = new FileOutputStream (tempFile);
out.write(content);
out.close();
/* Если мы не можем переместить файл в другую файловую систему, нужно его
скопировать и переименовать. Ищите причину в документации к API: (http://
docs.oracle.com/javase/7/docs/api/java/io/File.html#renameTo(java.io.File)):
"Многие аспекты работы этого метода имплицитно зависят от платформы: опе-
рация переименования может и не привести к переносу файла из одной систе-
мы в другую — файл может быть достаточно большим, и этого может и не слу-
читься, если файл с таким именем пути уже существует. Выводное значение
всегда должно проверяться, чтобы была твердая уверенность в необходимости
переименования".*/
if (!tempFile.renameTo(outFile)) {
    FileInputStream source = new FileInputStream(tempFile);
    FileOutputStream destination = new FileOutputStream(outFile);
    FileChannel sourceFileChannel = source.getChannel();
    FileChannel destinationFileChannel = destination.getChannel();
    sourceFileChannel.transferTo(0, sourceFileChannel.size(),
    destinationFileChannel);
}
```

---

1 www.hulver.com/scoop/poll/1086869940_ZZXTfTNN.

2 Этот пример значительно упрощен, в частности, здесь отсутствует какая бы то ни было обработка ошибок. Кроме того, было бы более целесообразно использовать библиотеку наподобие FileUtils из Apache Commons, в которой автоматически учитываются проблемы с переименованием. Однако в таком случае здесь не было бы этого примера.

Здесь происходит следующее: наша программа должна заполнить файл результатами измерений с интернет-сервера. При этом в случае потери связи не должна быть записана лишь половина данных, а должен работать принцип «всё или ничего». Поэтому мы записываем данные (`content`) сразу же во временный файл. Поскольку мы следуем общепринятым правилам, то помещаем эти файлы в каталог для временных файлов (`temp`-каталог). Запись прошла успешно, поэтому мы уже уверены в том, что можем спокойно пользоваться временным файлом.

Мы хотим использовать имеющийся в Java метод `renameTo()`, чтобы временный файл переместить в конечное место его использования. Мы ожидали, что это не вызовет трудностей (в папке назначения уже имеется хранилище с таким именем).

К сожалению, позже выясняется, что Java позволяет использовать для переноса и переименования данных лишь собственную системную функцию, а ее структурная особенность, в частности в ОС Linux, не позволяет переносить файлы за пределы некой файловой системы, например с одного жесткого диска на другой. Поэтому этот вариант функции в Java претерпевает крах в случае, если каталог `temp` находится на другом жестком диске, что вполне естественно. Разработчики метода `renameTo()` опрометчиво решили использовать возвращаемую переменную типа Boolean, которая будет показывать, успешно ли прошла операция, без указания причины сбоя при неудаче (это может произойти лишь в некоем исключительном случае).

Не беда — в таком случае мы скопируем данные в новое место (в примере мы опустили некоторые случаи сбоев и способы их устранения) и затем сможем удалить временный файл. Это займет больше времени, но приведет нас к цели. Поскольку мы можем очень скоро забыть об описанном ограничении метода `renameTo()`, будет разумно написать комментарий, который сможет и через два года объяснить программисту, почему нужно проверять успешность выполнения `renameTo()` и использовать другой способ переноса файлов в конечное хранилище в случае появления ошибок при использовании последнего.

Если вы видите необходимость в изменениях, но сейчас не можете сконцентрироваться на этом или у вас недостаточно времени

Даже один лишь факт необходимости переделать или переписать участок кода требует комментария. Если известно, что нужно переписать что-то, но на это не хватает ресурсов, поскольку все внимание уже отдано другой проблеме, поможет хотя бы комментарий типа «TODO: этот код написан плохо, переписать», который сможет послужить определенной пометкой. Информация о том, что автор при написании кода был не уверен в нем, лучше, чем отсутствие информации. А еще лучше будет, безусловно, если он напишет, почему не уверен.

Приведенный далее комментарий плохой (он взят из движка блога Riesenmaschine). Тем не менее его написание нельзя охарактеризовать как совершенно бесполезную трату времени:

```
# TODO: тут уже есть некоторые недочеты
```

Возможно, автор комментария знала на момент его написания, что же пошло не так, но при этом она просто упомянула о проблеме и не посчитала важным указать

какие-либо подробности. Если вы поняли, что не так, и даже уже предполагаете, как можно это исправить, обязательно напишите об этом в комментарии! То, что в данный момент кажется вам элементарным, в действительности кажется таковым только потому, что вы прямо сейчас погружены в эту тему.

Определитесь с универсальным обозначением проблемных мест в коде, не пишите одновременно `to do`, `TODO`, `FIXME`, `XXX` и `!!!!!` — в дальнейшем это усложнит их поиск. По общепринятому правилу ярлыком `TODO` обозначаются все места, которые еще поддаются какому-то исправлению. `FIXME` обозначает, что здесь уже все потеряно, а вот `XXX` обозначаются места, к которым, как к проводам с высоким напряжением, лучше не подходить. В различных проектах принято устранять имеющиеся `TODO` и `FIXME` до следующего релиза.

КОММЕНТАРИИ TODO

Во многих средах разработки с помощью TODO обозначаются те комментарии, которые позже переходят в разряд задач. Есть даже специальный вид в редакторе, который предоставляет перечень всех комментариев TODO, а при компиляции эти пункты перечисляются как проблемные классы.

Если код предназначен для ограниченной сферы использования

Биоинформатик из университета штата Калифорния Аарон Дарлинг рассказывает журналу *Nature* о таком случае, вызванном небрежным ведением документации: код, написанный им для сравнения геномов, был предназначен только для работы с близкородственными организмами. Поскольку он нигде четко не упомянул об этом ограничении, другие исследователи использовали его код для сравнения организмов — дальних родственников, что привело к невообразимым результатам: «Хорошо, что мне это бросилось в глаза, потому что результаты исследований других ученых были совершенно неверны, но они даже не могли подумать об этом, потому что я плохо сопроводил документацией свой код»[1].

Если вы временно превращаете части кода в комментарий

Часто в целях тестирования некую часть кода, которая, например, проверяет, имеет ли пользователь достаточно прав на запуск какой-либо функции, помещают в комментарий. Это быстрый и удобный способ отладки отдельных частей кода[2] (см. главу 16). Но как только вас что-то отвлекает, возникает угроза, что эта часть кода так и останется комментарием (`/* if (not user_is_authorized) return*/'`). Если помечать такие места с помощью `XXX`, их легче будет потом найти. А еще лучше указать причину превращения кода в комментарий, потому что, кто знает, может, программист и не вернется к этому месту на протяжении всего года. Все это стоит отметить и для временно измененных значений: `$days_to_keep = 1; // CHANGEME` не так уместно, как `$days_to_keep = 1; // XXX поменять обратно на 7`.

---

1 www.nature.com/nature/journal/v467/n7317/index.html.

2 Это также свидетельствует и о плохом проектировании. Лучше все-таки было бы подумать о коде с точки зрения тестирования и написать соответствующие сценарии для тестирования.

Если вы уже попробовали первый приходящий в голову способ решения проблемы, но потерпели неудачу

Мысль, которая сегодня вам кажется банальной, и через три месяца, возможно, будет казаться такой же, поэтому вы и во второй раз можете остаться ни с чем, потеряв уйму времени. Поставьте знак комментария перед кодом и напишите что-то вроде:

```
// Это не работает (по непонятным причинам)
```

Если вы не последовали нашему отличному совету использовать систему контроля версий

В таком случае будет полезно указывать даты для любых произведенных изменений. Если над кодом работают несколько человек, следует также вставлять сокращенные имена. Так у вас появляется возможность через три недели после того, как все пошло наперекосяк, догадаться о причине происходящего. Это особенно важно применять к изменениям, которые вы не совсем осознаете или считаете неуместными. Недостаток: вы не сможете привести в систему все ваши пометки после того, как поменяете код. Подумайте еще раз над тем, чтобы начать пользоваться системой контроля версий.

Если решение может показаться программисту, читающему код, слишком сложным

Если вы действительно пришли к какому-то заумному решению, обоснуйте его целесообразность («Так код станет работать на 75 % быстрее») и докажите, что оно действительно кардинально отличается от других. Когда через некоторое время вы в следующий раз посмотрите на код, причина может отпасть за ненадобностью. А если вы оставите свое решение без комментария, тут же появится кто-то, кто сразу признает код нечитабельным и разнесет его в пух и прах.

Если вы уже знаете или предполагаете, что что-то делаете неправильно

Если вы, к примеру, не тестировали тот или иной участок кода на сбои, а просто по каким-то косвенным причинам знаете или предполагаете о них, не нужно стыдливо умалчивать об этом. Напишите честный комментарий и пометьте его `TODO`. Если позже возникнут проблемы, вы сможете быстрее найти их причину.

Если вы часто страдаете от путаницы из-за множества сложных связей в управляющих структурах

Можно взять за правило в конце таких структур оставлять комментарий:

```
for (cat_counter = 0; cat_counter < count(cats); cat_counter++) {
    if (cat_counter % 2 != 0) {
        for (hedgehog_counter = 0; hedgehog_counter < count(hedgehogs);
        hedgehog_ counter++) {

            // много кода

        } // конец hedgehog_counter
```

```
    } else {

        // еще больше кода

    } // конец cat_counter
} // конец for cat_counter
```

Вообще в дальнейшем лучше стараться использовать более краткие, удобные для восприятия управляющие структуры. Несколько советов, как это можно сделать, вы найдете в главе 15.

Если приходится иметь дело с обрывками чужого кода

Указывайте источник чужого кода. Возможно, по поводу него позже возникнут вопросы или автор однажды представит новую, улучшенную версию кода. (Помните: это касается только обрывков кода. Если вы используете большие участки чужого кода, то есть от нескольких строк до целой библиотеки, следует сохранять чужой код в отдельном файле, и ни в коем случае нельзя вносить в него изменения.)

Если вдруг в середине кода появляются числа

Надеемся, что после прочтения и усвоения главы 14 в вашем коде вдруг из ниоткуда не появятся числа. Вы им присвоили константы. Определение константы почти всегда принесет еще больше пользы, если вы добавите комментарий, в котором объясните, почему выбрали именно это значение и для чего оно используется.

Даже если причина будет звучать следующим образом:

```
// найдено методом проб и ошибок: это значение вроде охватывает
// наименее губительные для сетчатки глаза цвета
```

вы все равно облегчите жизнь программистам, читающим код.

Проблемные комментарии

Комментарии могут наносить вред коду, в частности, тогда, когда они неверны и ведут программистов по ложному следу. Даже разумные комментарии влекут за собой дополнительную работу: в будущем придется переделать не только код, а еще и относящийся к нему комментарий. Если есть уверенность в ненужности какой-то информации, стоит задать себе вопрос: «Могу ли я перенести эту информацию из комментария в код?»

Ранее мы рекомендовали, в основном новичкам в программировании, писать больше комментариев, чем кажется необходимым. Если следовать этой рекомендации, нужно помнить, что еще важнее обдуманно удалять комментарии, если они повторяются или не соответствуют действительности. Многие разработчики неохотно стирают код, потому что с его написанием связано много усилий. Такое отношение к работе переносится и на комментарии, но это в корне неправильно.

Повторяющиеся комментарии, которые описывают вчерашнее состояние кода, еще хуже, чем отсутствие комментариев. Если у вас нет времени обновить комментарий, хотя бы пометьте его `TODO` и кратко поясните, почему он устарел. Если он приводит к избыточности, сотрите его. Хорошие программисты достаточно бесцеремонны в удалении кода, и еще больше — в удалении комментариев, поскольку они знают, что при необходимости всегда могут все вернуть назад — например, использовать систему контроля версий или без особых усилий написать все заново.

Вот еще некоторые признаки проблемных комментариев.

Комментарий слишком длинный

Каждая не совсем банальная функция должна описываться вначале одним-двумя предложениями. Если нужно больше, с функцией явно что-то не то. Возможно, она хочет все и сразу или слишком сложно задана.

Комментарий ссылается на переименованные элементы

С помощью инструмента рефакторинга вы поменяли имя класса, функции или переменной, но используемый вами инструмент не учел комментарии. В комментарии теперь стоит устаревшее имя элемента. Само по себе игнорирование комментариев инструментом рефакторинга не является неправильным, поскольку могло произойти так, что вы небрежно назвали какую-то переменную `variable`, а потом возьмете и назовете ее `timestamp`. Это вызовет большую путаницу, если заменить все упоминания `variable` на `timestamp` в комментариях. Во многих средах разработки существует функция пересмотра комментариев во время переименования переменных.

Комментарий содержит сокращения

Здесь действует такой же принцип, как и для имен переменных (см. главу 5): применять сокращения только тогда, когда они используются более часто, чем полные слова (например, HTML, PDF).

Комментарий, который лучше было бы поместить в commit message системы контроля версий, или наоборот

Если вы исправили баг и применяете систему контроля версий, перед вами встает вопрос: куда лучше поместить комментарий: в код или в *commit message*? Как правило, производя commit, имеет смысл лишь пояснить, *что* было изменено. Хорошо координируемые проекты используют систему управления задачами, например Jira, и привязывают *commit message* к соответствующему ID задачи для того, чтобы позже можно было отследить причину изменений. Если же речь идет не о хорошо развитом проекте, можно самому указать причину изменений в комментарии. Это не так удобно, потому что длинные истории не добавляют коду читабельности.

Комментарий непонятен программистам, для которых предназначен

Очень тщательно обдумывайте, будет ли понятен комментарий, в случае, если код будет читаться непрограммистами или дилетантами. Например, такое может

произойти, когда вы всему миру показываете плагин для WordPress, новое дополнение для браузера или инструмент Greasemonkey или когда пользователи вашего кода перед запуском должны ввести в файл с кодом некие данные для входа.

Оригинальные примеры искусства комментирования

```
# потрачено_часов_впустую = 16
# Одному богу известно
# посвящается моей жене Дарлин
# пьян, починю позже
# временное, черт его побери!
# работай, ну пожалуйста!
# стремно удалять
# Обращение ко мне в будущем. Прости меня за все!
# тяжело было писать, так пусть тяжело будет и читать
# TODO: Почини. Что починить?
# Ruby lexer адаптирован из irb — внутренние компоненты не описываю, потому что боюсь.
# Этот код может любить одна лишь мать.
# Что-о-о-о-о, почему он не запускается?
```

Источники: stackoverflow.com/questions/1 84618/what-is-the-best-comment-in-source-code- you have-ever-encountered/, blog.fefe.de/?ts=b27 08668, @inoxio / Twitter, @IAmMarth / Twitter.

7 Чтение кода

> Читать код — это иногда то же, что искать правду внутри себя в наше время.
>
> *Рин Ройбер/@rinpaku, Twitter, 10 мая 2013 года*

В большинстве случаев код намного легче писать, чем читать. Это связано с тем, что при написании кода у нас уже есть образная модель программы в голове — ее нужно лишь записать. Это «лишь записать» уже представляется довольно сложным делом. Однако для того, чтобы код, особенно чужой, прочитать и понять, нужно в процессе чтения суметь восстановить образную модель программы, существовавшую в голове разработчика данного продукта, а это намного сложнее. Даже читая качественный код, нужно сразу улавливать особенности его синтаксиса и при этом постоянно помнить о поведении кода как единого целого. Это следует учитывать, если вы пытаетесь узнать подробности работы некой функции. Это сложно не только для новичков.

Тем не менее периодически читать код других программистов очень важно, поскольку это помогает стать хорошим программистом. Довольно часто другие программисты по-другому и лучше справляются с некоторыми задачами. Поскольку у нас во время работы над каким-то вопросом не всегда есть возможность проконсультироваться с коллегами, а может быть, нам не хватает на это смелости, это хорошая альтернатива узнать о других вариантах решения проблем.

Думаете, это действительно того стоит?

Вполне естественно не хотеть читать код других. Мозг большинства людей противится чрезмерной работе и в целом предпочитает расслабление усердному труду. Зачем читать и пытаться понять неразборчивый код, если вместо этого можно посмотреть изображения котят или посмеяться над кучей мемов? Автор блога *Coding Horror* Джефф Атвуд пишет: «Никто не читает чужой код ради удовольствия — да я и свой не всегда читаю с большой охотой. Смешно предполагать, что кто-то устроится в кожаном кресле и будет наслаждаться бренди за чтением чужого исходника»[1].

---

[1] www.codinghorror.com/blog/2012/04/learn-to-read-the-source-luke.html.

И все же найдутся такие программисты, которые захотят поспорить с Джеффом, отрицая комичность предложенной им ситуации, и даже получат истинное удовольствие от чтения «утонченного» кода. Возможно, вы придерживаетесь точки зрения Джеффа Атвуда. Может, вам ближе мнение второй группы и даже когда-нибудь вы сами с упоением будете читать код. Но для начала попробуйте заключить следующую сделку со своим мозгом: вы будете время от времени читать длинные куски кода и обдумывать прочитанное, не отвлекаясь на другие, более приятные дела.

Если же вы испытываете отвращение при мысли о чтении кода, будьте откровенны с собой: нет, так происходит не из-за кода. Это никак не связано с вашим интеллектом. Это связано лишь с тем, что вам нужно еще и думать над чужим кодом. Это нормально, поскольку мышление признает необходимость этой работы, хоть ее можно все время отрицать. Но это всегда нужно делать, обосновывая не тем, что «этот код невыносим» или «я еще так глуп», а тем, что «я не буду прямо сейчас об этом думать».

Полное нежелание прочесть даже самый малый фрагмент кода может препятствовать вашему самообразованию, если вдруг вы захотите что-то узнать. Можно рассматривать библиотеку, которую хотелось бы применить в своем коде, или читать познавательную, но переполненную примерами кода книгу, однако это будет выглядеть как бесполезное пролистывание страниц, вызывающее отторжение, если у вас недостаточно практики в чтении кода.

Если вы захотели начать работу с новыми языком или технологией, вам очень поможет чтение существующего кода, и это позволит получить некое представление о том, в чем заключаются используемые в данном языке или технологии стандарты. Здесь не нужно в совершенстве понимать код — зачастую поверхностное знание будет вам на руку. Если вы уже знаете язык или функции библиотеки до такой степени, что можете механически подражать тому, как другие их используют, значит, вы знаете достаточно для того, чтобы сделать первые неуклюжие шаги в новой сфере. Остальное постигается методом проб и ошибок, а также дальнейшим чтением.

Вы также можете принудить себя к чтению чужого кода с помощью неких внешних обстоятельств, в таком случае вам поможет уже имеющийся опыт чтения кода. Для штатных программистов, как и для многих специалистов, не имеющих прямого отношения к программированию, но время от времени сталкивающихся в работе со скриптами и программами, также свойственно анализировать код предшественников или коллег. Хорошего анализа не выйдет, если предварительно не прочитать код и не понять его хотя бы в общих чертах.

Даже если вы программируете только ради своего удовольствия и для себя, вы можете оказаться в ситуации, когда необходимо прочитать чужой код. В настоящее время простые программы зачастую создаются на основе уже написанного кода. Языки имеют собственные стандартные библиотеки, а под прикладную среду, такую как Web или мобильные устройства, едва ли можно программировать без использования фреймворка. Все большее количество готового кода становится Open Source, что позволяет заглянуть в исходники. Открытый исходный код часто поставляется не с лучшей документацией, хотя PHP и jQuery представляют собой два впечатляющих исключения в этом плане. Наконец, если вы никак не можете

устранить ошибку и спрашиваете о ней на форумах у других программистов, их ответ будет содержать код, который вам нужно будет как-то понять.

Даже если вы написали код, которым будете пользоваться не только сами, и это произошло всего лишь после трех недель чтения книги «Построение роботов из LEGO и программирование», вам все равно не избежать чтения чужого кода. А именно это происходит в том случае, если ваш код не выполняет тех операций, которые он должен выполнить. В такие моменты вы сами отвечаете за все компоненты кода, даже за те, которые вы сами не писали. Вы не можете искать отговорки в том, что не отвечаете за тот или иной фрагмент кода. В тот момент, когда вы показываете свой код всему миру, вы становитесь ответственным за все ошибки в нем. Это так же, как и при рождении детей: можно от них отказаться, но это будет, мягко говоря, непорядочно.

Часто, читая чужой код, можно научиться лучше следить за читабельностью собственного кода, поскольку вы уже знакомы с более или менее понятным кодом и побывали в шкуре того, кому нужно следовать чужому мышлению. А тот, кто однажды проклинал плохие примеры кода, непонятные отступы и устаревшую документацию, будет, вероятно, больше задумываться о легкости чтения собственного кода. Дуглас Крокфорд говорит: «Сегодня читаемость кода — мой главный приоритет. Она важнее эффективности и почти так же важна, как корректность, и я думаю, что читаемость кода — важнейший шаг к его корректности»[1]. Для начинающих программистов это хорошая новость, поскольку писать читаемый код легче, чем высокооптимизированный.

А теперь предлагаем наши рекомендации для эффективного чтения чужого и собственного кода.

Сперва читайте документацию

> Час чтения кода может сэкономить минуту чтения документации.
>
> *Диомидис Спинеллис. Чтение кода*

Если у программы или исходника, с которыми вы работаете, есть документация, прочитайте сначала ее. Это сэкономит вам довольно много времени при чтении кода, поскольку документация содержит обзор функций программы, а также отвечает на вопрос о том, как программу обычно используют. В противном случае вам придется усердно вылавливать это из самого кода.

Особенно в тех случаях, когда в код встроены неизвестные библиотеки, для ознакомления с ними часто достаточно прочесть документацию. В исходном коде можно лишь заплутать (код библиотеки часто внушительных размеров) и быстро уйти от цели именно применить библиотеку. Библиотеки — это, собственно, и есть код, предназначенный для дальнейшего использования в других проектах. Он должен представлять собой интерфейс и не требовать от пользователя (то есть программиста) лезть в дебри. Библиотеки обычно снабжены документацией.

---

[1] *Сейбел П.* Кодеры за работой. — М.: Символ-Плюс, 2011. — С. 107.

То, что не очень хорошо распознается при чтении документации, — взаимосвязь: как можно разумно использовать выводимое значение одной функции для запуска другой, какие лучшие практики имеются в языке, как можно встроить библиотеку в исполняемую программу? Ответы на эти вопросы вы найдете при ознакомлении с образцами кода, которые обычно предоставляются разработчиком языка или библиотеки. Для обсуждения наиболее известных программ и библиотек в Сети существуют форумы, записи в блогах и/или фрагменты кода (сниппеты). Если вы не можете найти вообще ничего, что служило бы документацией о применении кода, это довольно подозрительно — либо библиотека была недавно разработана и поэтому еще недостаточно надежна, либо никто ею не пользуется. Это не зазорно, однако неудобно для вас, поскольку в таком случае придется время от времени просить помощи у других пользователей.

У начинающих в этом плане есть преимущество: им не надо думать, плох или хорош код, который другие пользователи публикуют на форумах. Он наверняка не хуже их собственного кода. Если вы полагаетесь не только на код одного человека, через некоторое время вы станете опытнее, даже если будете перенимать у коллег дурные привычки или не лучшие решения проблем.

Распечатайте исходник

Если вам удобнее работать с бумагой, распечатанный код будет более наглядным, чем экранная версия. На бумаге быстрее пишутся комментарии, некоторым легче читать распечатанный вариант, и можно положить рядом две страницы, пока вы редактируете код с помощью компьютера. Не нужно думать, что это дурно, даже опытные программисты так делают: «Если же части программы тесно связаны между собой и надо читать ее целиком, чтобы понять отдельные элементы, — это просто кошмар. [...] Я распечатываю все, сажусь на пол, раскладываю вокруг листы распечатки и делаю на них пометки»[1].

Этот метод интересен тогда, когда вы не работаете с интегрированной средой разработки, поскольку современные среды разработки во многом помогают понять код без необходимости использования печатной копии. Кроме того, часто длинные имена настолько раздувают текст, что он не помещается на странице, а из-за переносов нарушается структура документа. В крупных проектах печать кода на бумаге в любом случае становится не самым лучшим выходом из проблемной ситуации: если вы распечатываете большую библиотеку, то получаете документ из 500 страниц. Это довольно затратно, к тому же вы просто не сориентируетесь в таком огромном объеме. Поэтому следует распечатывать не все страницы, а только наиболее необходимые, например дерево каталогов проекта — так у вас перед глазами всегда будет перечень имеющихся файлов. Или перечень API, который поможет знать, какие функции вы можете использовать. Если вы хотите прочесть программу, а не библиотеку, распечатайте начало программы (`main()` в языках типа C, а именно PHP, Java, JavaScript, C++ или C#) и функции, которые кажутся вам наиболее интересными.

---

[1] *Сейбел П.* Кодеры за работой. — М.: Символ-Плюс, 2011. — С. 172.

Схематично изобразите для себя, для чего предназначены некоторые компоненты программы

Восьмидесятые годы были периодом расцвета схематического представления программ. Тогда для визуализации работы программы часто применялись структограммы[1] и блок-схемы[2]. Сегодня такие варианты представления программ немного потеряли свою популярность. Тем не менее они дают отличную возможность решить проблему путаницы в коде — обеспечивают менее абстрактное видение программы без таких элементов синтаксиса, как `{` или `]`.

Будет вполне достаточно взять один лист бумаги и расписать программные процессы с помощью блоков, стрелок и кругов — неважно, какие именно фигуры станут обозначать элементы программы, поскольку такое графическое представление необходимо лишь для того, чтобы вы поняли ее. Если у вас достаточно времени или вам это нужно для выполнения более важных заданий, можете создать такую блок-схему с помощью языка моделирования UML и таким способом более профессионально представить статистические и динамические структуры программ.

Кроме того, уже предпринимались попытки разработки инструментов визуализации, преобразующих код в более понятную форму. Не спешите доставать фломастеры, а попробуйте поискать такой инструмент по ключевым словам `code visualization`, указав язык программирования, с которым работаете. Если вы изучаете код высококлассного проекта, вполне вероятно, что наглядные диаграммы уже существуют и вам остается их только отыскать.

Сверху вниз, от простого к сложному

Если документации недостаточно или ее вовсе нет, вам следует как можно скорее самому получить общее представление о структуре кода. Сколько файлов он содержит? Что говорят имена последних о функциях программы?

Файл, который называется так же, как проект, в большинстве случаев содержит начало кода программы и ее важнейшие компоненты. Здесь скрываются методы типа `main()` или, в случае библиотеки, функции, которые вам, как их пользователю, будут наиболее интересны. В первую очередь вам следовало бы изучить файлы такого типа, а все те, в имени которых содержится `helper` или `util`, не заслуживают вашего внимания.

Текстовый редактор с поддержкой сворачивания очень помогает понять общую структуру модуля кода. Сворачивание — это возможность временно скрыть части кода таким образом, что, например, будут видны только внешние уровни отступа. Статья «Википедии» «Сворачивание» содержит обзор редакторов и сред разработки, поддерживающих такую возможность.

---

1 en.wikipedia.org/wiki/Nassi–Shneiderman_diagram.
2 en.wikipedia.org/wiki/Flowchart.

Если вы уже ознакомились с кодом в общем, пришло время вникать в детали. Начните с подробного чтения понятных фрагментов, иначе у вас пропадет всякое желание читать и вы просто брезгливо бросите файл с исходником в дальний угол. Даже если вы справились со сложными фрагментами кода и разобрались в их функциях, это вам мало что даст в плане цельного понимания сущности кода. Поэтому лучше посвятить драгоценное время поверхностному чтению огромных фрагментов кода.

Если вы видите, что застряли в каком-то месте и не можете продвинуться дальше, лучше отвлечься на что-то. Упрямство разработчика в определенных ситуациях имеет положительный эффект, но лучше им не злоупотреблять. По возможности постарайтесь позже вернуться к тому или иному модулю или функции в другом настроении и с новыми знаниями — это поможет вам во всем разобраться самому.

Учитесь искусству следопыта

Вывод отладочной информации, запись в лог-файл и/или assert'ы обычно можно обнаружить в важных местах или потенциальных источниках ошибок. Пометки FIXME и TODO обычно находятся во фрагментах некачественного кода. Используйте информацию, заключающуюся в именах переменных и функций, — если эти имена дают малые или ошибочные сведения, лучше обратить на них особое внимание.

Учитесь распознавать рекурсивные функции, поскольку они часто служат показателем того, что в данном случае обрабатываются иерархические данные, например дерево файлов:

```
function findCsvFiles (fileOrDirectory, csvFiles) {
    if (fileOrDirectory.isFile() && (fileOrDirectory.name.ends-
With('.csv'))) {
        return fileOrDirectory;
    } else if (fileOrDirectory.isDirectory()) {
        var files = fileOrDirectory.list();
        for (var file : files) {
            csvFiles.add(findCsvFiles (files, csvFiles));
        }
    }
}
```

Эта функция рекурсивная, поскольку в нижней части она вызывает саму себя, если в перечислении элементов каталога нашла подкаталог. Затем она просматривает дочерние папки данного каталога и добавляет найденные CSV-файлы в массив `csvFiles`. Если она находит еще одну папку уровнем ниже, то сначала обработает ее, а потом вернется к основному каталогу.

Научитесь распознавать, как в выбранном вами языке начинается асинхронная обработка данных (потоки выполнения и метод Ajax). Часто это может показывать, что время обработки может отличаться от того, которое предусмотрено линейным кодом:

```
var db_conn;
Thread.new ({
    var success = db_conn.connect (database: "mysql@server",
                  user: "johannes", password: "pass");
```

```
    if (success == false) {
        throw DatabaseConnectException.new();
    }
    ... // продолжение кода
});
print ("connecting to database");
```

Эта программа устанавливает соединение с базой данных в параллельно выполняющемся потоке. Поскольку в данном случае запускается асинхронная обработка, команда `print` вызывается *до того*, как программа в действительности соединилась с базой данных. У этого есть свои плюсы, поскольку интерфейс пользователя не зависает, пока программа устанавливает связь с базой данных. Однако при чтении кода нужно помнить, что слишком рано в строке с командой `print` писать `connected to database`, поскольку в случае сбоя связи с базой данных `DatabaseConnectException` срабатывает только после команды `print`.

Соотношение 80/20 оптимально (в большинстве случаев)

Несомненно, мы очень радуемся, если действительно поняли код, но для выполнения определенной задачи в программировании это понимание зачастую не нужно, а новичкам не всегда это и удается. Если вы читаете код для совершенствования собственного мастерства программиста, спокойно анализируйте вплоть до мельчайших деталей код, который вам будет казаться сложным. «Вцепиться» в код в таком случае бывает полезно, поскольку это принуждает мозг следовать за развитием событий в коде шаг за шагом — не имея даже приблизительного представления, особо не угадаешь, о чем именно идет речь.

Если же такая задача вам не по зубам или у вас недостаточно времени, не отчаивайтесь: приблизительного представления, как правило, достаточно, а в деталях вы разберетесь, когда будете непосредственно работать с кодом. Прочесть код и разобраться в отдельных его фрагментах, которые вы потом, скорее всего, будете использовать или переписывать, также может быть вполне разумным решением — так вы избавитесь от ненужной работы. Хотя у тщательного изучения кода есть свои бесспорные преимущества, нужно помнить, что, возможно, этот код больше вам никогда не повстречается.

Помните о форматах данных

Если вам приходится работать с макросом VBA[1], встроенным в файлы Excel, нужно рассматривать не только код. Данные из книг Excel часто позволяют быстрее, чем источники Visual Basic, понять, для чего лучше использовать код. То же самое верно и для проектов в Access, равно как и для других программ, которые активно обрабатывают данные.

---

[1] VBA — Visual Basic for Applications, классический скриптовый язык, используемый в крупных компаниях.

Научитесь распознавать определенные форматы данных, например:

- *20060712* — метка времени, то есть год, месяц, день (именно в такой очередности). Этот формат часто можно встретить в полях дат в базах данных SQL;
- *1152680400* — десятизначные числа, начинающиеся с чисел от 10 до 14, часто являются временными метками Unix, измеряемыми в секундах, с отсчетом от 1970 года;
- *192.168.1.10* — четыре числа, разделяемые точками, имеющие значение от 0 до 255, обозначают IPv4-адреса;
- *2001:DB8:0:0:211:22FF:FE33:4455* — восемь чисел, разделенных двоеточием, или меньше чисел, но с одним двойным двоеточием в ряду (::), обозначают IPv6-адреса;
- *f09090* — шесть шестнадцатеричных цифр часто обозначают цвета.

Истина в самой программе

Если вы совсем не понимаете, в чем проблема, и/или не знакомы с используемыми технологиями, не факт, что вы сразу же все поймете после прочтения кода — программы сложны и зачастую к тому же громоздки. В конце концов, часто вас спасут лишь запуск программы и наблюдение за ее работой.

Если у вас уже и так имеется исходный код, можете воспользоваться случаем и удалить комментарии в определенных фрагментах или вставить команду `print()`, чтобы лучше понять принципы работы программы и ее структуру. Если отладчик для вас не неведома зверушка (см. подраздел «Отладчики» раздела «Инструменты и стратегии диагностики» главы 13), вы еще быстрее придете к своей цели.

Хорошая возможность узнать о функции кода более подробно заключается в тестировании. Если вы читаете код, который еще все равно не имеет тестовой документации, вы можете сами написать модульные тесты (см. главу 16), чтобы связать собственное представление о коде с реальностью.

Коллективное чтение кода

Если у вас есть коллеги-программисты или вы программируете в свободное время и ваши друзья тоже этим занимаются, можете значительно улучшить собственные профессиональные навыки посредством совместного чтения кода. Если вы доверяете друг другу, проанализируйте собственный код совместно с друзьями или коллегами. Если это кажется вам кошмаром, читайте вместе чужой код.

Цель чтения кода не заключается в том, чтобы определить сильнейшего и найти больше всего ошибок в чужом коде. Главное здесь — понять код. Возможно, читая код, вы не полностью поймете его функцию или в чем-то ошибетесь. Если вы обсуждаете код с другими людьми, то быстро становится заметно, в каких местах ваши предположения о выполняемой кодом функции расходятся.

Если же ну совсем ничего не помогает, у вас есть полное право попросить помощи у других (но до этого — ни-ни!). О том, как этого добиться — постоянно действовать им на нервы или тратить их время, — речь пойдет в главе 8.

8 Где искать помощь

> Kritical: Кристина, вы должны научиться самостоятельно разбираться в проблеме.
>
> Кристина: Как мне это сделать?
>
> *www.bash.org/?3936*

Если на сегодняшний день уровень программирования некоторых авторов этой книги колеблется от среднего до хорошего, то этим они обязаны в том числе и тем добрым людям, которые помогали им разобраться в базовых вопросах. На рассвете эры программирования считалось большой удачей иметь пару-тройку знакомых, которые тоже программировали. Сегодня в Сети есть тысячи мест, где новички могут избавиться от мучающих их вопросов и — если они все сделают правильно — даже получить на них конструктивные ответы. Если вопрос задан неверно, то, конечно, может случиться и так, что от вас отделаются резкими замечаниями в духе «сначала поищи, а потом спрашивай», lmgtfy[1] или RTFM[2]. Много злобы и гнева приходит в этот мир из-за того, что программисты неудачно формулируют свои вопросы. Давайте представим себе такого программиста, который оставил на форуме для разработчиков библиотеки PHPschnargl (вымышленной) следующий крик о помощи:

```
от кого: achim1990
кому: phpschnargl-dev
тема: ПОМОГИТЕ!!!
В своем последнем проекте я пытаюсь использовать PHPschnargl, и ничего не
выходит. Когда я захожу на страницу, мой сценарий перестает работать.
Пожалуйста, подскажите, почему он не работает! Вот мой код
```

(Затем следуют два скриншота.)

Запрос такого вида ожидаемо вызовет раздражение, потому что автор допустил несколько ошибок.

- Он выбрал не тот форум. Он думал, что форум phpschnargl-dev будет правильным выбором, поскольку dev в названии указывает на девелоперов (разработчиков) как

---

[1] lmgtfy — сокращенное от Let me google it for you — «Давайте я поищу это для вас в Google». Часто ответ дается в виде ссылки на lmgtfy.com.

[2] RTFM — сокращенное от Read the fucking manual — «Читай чертову инструкцию!».

на целевую группу. И хотя так оно и есть, автор, к сожалению, упустил из виду тот факт, что это форум разработчиков самой библиотеки PHPschnargl. А как программист, который просто пользуется библиотекой, он должен был обратиться за помощью на форум для пользователей phpschnargl-user.

- Использовать «ПОМОГИТЕ!!!» в качестве заголовка темы совершенно бессмысленно. Любому, кто захочет помочь, придется сначала открыть сообщение, чтобы узнать, в чем именно требуется помощь.
- Описать проблему — не значит донести суть этой проблемы до кого-либо еще. Автор вопроса не предоставил никакой фоновой информации, то есть не указал версию PHP, версию библиотеки и т. д. Полезно было бы также прикрепить журналы ошибок, ведь опытный программист может получить из них наиболее важную информацию.
- Код очень длинный. Мало кто может с легкостью вникать в чужой код. И вряд ли кто-то будет напрягаться, если код не сокращен до основной проблемы.

Даже простые запросы о помощи не так-то просто составить, как можно было бы предположить. В этой главе содержится основная информация о том, как просить о помощи с максимальным для вас результатом и минимальными затратами нервов окружающих.

Подходящий момент

Основное правило гласит: уважайте чужое время, но и собственное попусту не тратьте. Тот, кто задает вопрос слишком рано, кажется несамостоятельным и крадет время других людей. Тот, кто спрашивает слишком поздно, тратит, возможно, совершенно бессмысленно шесть часов на проблему, которую можно было бы решить в течение нескольких минут с помощью одного простого вопроса. К тому же в Сети и, вероятно, во всем остальном мире полно людей, которым действительно нравится разбираться с чужими проблемами — при условии, что те, кому требуется помощь, до этого хотя бы попытались разобраться в вопросе самостоятельно.

Предварительная самостоятельная работа является обязательным проявлением вежливости и поможет вам выбрать правильное время для того, чтобы задать свой вопрос. Если на все пункты из следующего перечня вы можете ответить «Да», то, не стесняясь, просите о помощи.

Искали ли вы решение проблемы или, еще лучше, текст сообщения об ошибке через поисковик?

Пару советов по поиску вы найдете во врезке «Сначала ищем, потом спрашиваем». Если по какой-то причине вам пришло сообщение об ошибке на русском языке, выяснили ли вы, как это звучит на английском, и искали ли эту версию? Кстати, таким образом можно избежать ситуаций, когда вы, сидя перед монитором, сгораете от стыда, потому что кто-то опубликовал ссылку в ответ на ваш вопрос, не преминув отметить, что это самый первый результат в поисковике.

Если ваш вопрос связан с HTML, CSS, JavaScript или XML, вы воспользовались соответствующим валидатором для проверки и можете гарантировать, что ваш код соответствует всем стандартам?

Список часто используемых для веб-технологий валидаторов см. в главе 13. После проверки валидатор выдаст вам множество подробных сообщений об ошибках и несоответствиях. Одно из этих предупреждений, возможно, связано непосредственно с вашей проблемой. Пока ваш код не прошел валидацию, вы крадете своим вопросом время других людей. Впрочем, программы проверки заранее встроены в интегрированные среды разработки для популярных языков и форматов (см. раздел «Среды разработки» главы 20).

Поставили ли вы на максимум все настройки своего языка программирования, отвечающие за выдачу ошибок и предупреждений? Вы учли полученную информацию и исправили ошибки?

Подробнее данный вопрос рассматривается в главе 13.

Вы выяснили, есть ли официальный FAQ для программного обеспечения, библиотеки или проекта, по которым у вас возник вопрос? Вы искали там ответ на свой вопрос?

Это позволит избежать неловкого ощущения, которое возникает, если через полминуты после публикации вашего вопроса приходит ответ, который ссылается на пункт 2 официального FAQ.

Вы просматривали архив сайта, на котором хотите задать свой вопрос?

Возможно, этот вопрос задают постоянно, но ответственные за FAQ были просто слишком ленивы, чтобы включить его в список часто задаваемых вопросов.

Нет прощения тому, кто задал свой вопрос, не проверив все вышеперечисленные пункты. А тот, кто начинает свой запрос со слов: «Извините, еще нигде не смотрел, но это СРОЧНО» или «Я уже знаю, что код невалиден, но это не моя вина, а...» — заслуживает того, чтобы его номер оказался в телефонных справочниках всех продавцов России и чтобы они звонили ему каждый день в пять часов утра и начинали разговор со слов: «Я знаю, рановато для звонков, но мне удобно именно сейчас!» Кроме того, на подобный вопрос, вероятно, никто и не станет отвечать.

Вот как это работает: сначала ищем, потом спрашиваем

Ищите на английском языке, а не на русском

В мире значительно больше людей, которые обсуждают проблемы программирования на английском языке, чем тех, кто делает это на других языках. Соответственно, на английском у вас гораздо больше шансов найти ответ, особенно если проблема специфическая. Если же вы понятия не имеете, как звучат ключевые слова по вашей проблеме на английском языке, то, вероятно, поможет поиск статей по этой теме в русской «Википедии», а затем переход на англоязычную версию сайта.

Ищите ответ, а не вопрос

Поиск по запросу «Как мне прервать поток в Java» выдаст в первую очередь страницы, где есть такой же вопрос, а не ответ на него, а запрос «Прерывание потока в Java с бóльшей вероятностью выведет на страницу, содержащую ответ. Попробуйте угадать, какие формулировки могут использоваться в ответе, и ищите по ним. На практике это может означать, что вам придется сперва перебрать несколько вариантов, прежде чем выяснится, по каким ключевым словам вообще нужно искать, чтобы найти ответ. Подбирайте варианты с умом. Это одновременно и обучающий процесс, ведь после этого вы станете лучше разбираться в основных понятиях по вашей теме.

В случае конкретных проблем обычные форумы практически бесполезны

Если из поисковой системы вы перешли на страницу, уже по одному виду которой понятно, что это форум типа phpBB или vBulletin, можете сэкономить свое время и сразу ее закрыть. Кроме того, такие форумы зачастую индексируются поисковыми системами необоснованно, ведь ссылки на них уже давно ведут к совершенно другой части обсуждения. В редких случаях, когда предварительный просмотр результатов в поисковике содержит конкретное решение вашей проблемы, целесообразнее перейти по ссылке в кэш поисковой системы, который содержит именно то проиндексированное ранее место на форуме.

Не сдавайтесь сразу

Если результативность поиска оказалась не слишком высока, попробуйте задать вопрос в сочетании с такими наводящими ключевыми словами, как «вопросы и ответы», «решено», «решение», «пример кода», «ответил», «благодаря», «спасибо» или «справился».

Возможно, в Сети вообще нет ответа на ваш вопрос

Поскольку остальные неопытные программисты действуют по тому же принципу, что и вы, в Сети автоматически образуются места, где на никчемные вопросы даются никчемные ответы. Рассмотрите вариант приобретения «Кулинарной книги для гиков» по вашей теме или языку программирования из серии издательства O'Reilly; 80 % решений, которые вы ищете (или искали бы, если бы знали об их существовании), уже рассмотрены в этих книгах.

Возможно, все совсем не так, как вы думаете

Если вы, будучи не таким уж хорошим программистом, не нашли никакого решения своей проблемы, это не значит, что она такая новая и захватывающая и весь мир должен узнать о ней. Это значит, что ваша проблема либо не является таковой, либо вы составили себе неверное представление о ее характере. Лучше тщательно все обдумайте еще раз за чашечкой чая.

Спрашивайте в подходящем месте

После создания StackOverflow (stackoverflow.com) в 2008 году искать ответы на вопросы о программировании стало значительно легче. В отличие от прежде популярных форумов на этом сайте лучшие ответы могут двигаться вверх или вниз по ветке за счет выставленных оценок, так что вам не придется больше просматривать 32 страницы форума, чтобы в конце не найти никакого ответа. Эта система

сделала сайт популярным у хороших программистов. Шанс наткнуться там на полезный ответ настолько высок, что StackOverflow должен, пожалуй, стать вашим сайтом для поиска № 1. Разве что вам уже известен сайт, специализирующийся на вашей теме, на который вы возлагаете больше надежд. Кстати, из комментариев на StackOverflow видно, почему тому или иному вопросу был выставлен плюс или минус. Это поможет узнать много нового о том, как эффективно составлять такие запросы. Но с 2008 года прошло уже много времени, поэтому обратите внимание на дату, когда был дан найденный ответ. Некоторые проблемы программирования относительно вечны, ответы по другим могут стать неактуальными уже через пару месяцев.

Если StackOverflow не решил вашу проблему, попробуйте поискать ответ на Open Source. Если есть шикарная возможность написать на форуме для новичков, то вы, как начинающий программист, просто обязаны обратиться к этому ресурсу, ведь поначалу у вас возникнут, вероятно, тривиальные проблемы, с которыми на форумах для продвинутых вы будете только мешать. К тому же на форумах для начинающих не так резко осуждают за глупые ошибки.

К индивидуальным разработчикам с Open Source-Software можно обращаться с просьбой о помощи только в самом крайнем случае (читай: никогда). Единственное исключение из этого правила, если этот Open Source-проект настолько сомнительный и/или узкоспециализированный, что вы окажетесь одним из примерно семи его пользователей во всем мире и разработчик, вероятно, будет рад любому вопросу. И второе исключение — ваша фирма может выплатить разработчику гонорар за консультацию.

Структурируйте запрос правильно

Перед отправкой запроса решите для себя, о чем, собственно, вы хотите узнать. В случае конкретных ошибок все относительно легко. Но если вы не понимаете, например, как используется конкретная библиотека, то должны заранее продумать вопрос.

Он мог бы звучать так: «У кого-нибудь есть пример кода, как PHPschnargl версии PHP 5.0 инициализируется на Ubuntu Stoned Snail?» или так: «Инициализация PHPschnargl версии PHP 5.0 на Ubuntu Stoned Snail выдает ошибку e_param_range_violation. У кого-нибудь такое уже было? Можете сказать, что я делаю не так?»

В первом случае своим вопросом вы даете понять, что вам просто нужен рабочий код. Во втором — что вас интересуют сама проблема и ее решение. Оба варианта приемлемы, но на них могут быть даны разные ответы.

Запрос можно сформулировать, только если точно знаешь, что именно требуется от других людей. Нелишним будет сначала предоставить краткое описание проблемы. Оно должно содержать информацию о том, чего вы пытались добиться и что пошло не так. Для систем Ticket Tracking все время используются те же вопросы, что и при вызове экстренных служб, разве что обходится без пострадавших:

- Кто вы?
- Что вы сделали?
- Когда вы это сделали?
- Что случилось?
- Чего вы ожидали?

Этот список вы можете взять за образец, когда будете задавать вопрос интернет-сообществу. Таким образом читатели смогут быстро получить представление о проблеме и о том, смогут ли они помочь. Вместе с тем вы даете окружающим шанс указать вам на то, что ваш подход к решению проблемы является слишком сложным или просто неправильным.

Таким образом, лучше не пишите: «Я хочу вызвать php_schnargl.init с параметрами x и y, но получаю сообщение об ошибке e_param_range_violation», а напишите так: «Я хочу использовать функцию PHPschnargl для списка лиц с данными их аккаунтов в социальных сетях. Чтобы получить PersonalObject, профиль которого содержится в перечне URL, я вызываю функцию php_schnargl.init с помощью параметра x (пустой PersonalObject) и параметра y (URL-массив), но получаю сообщение об ошибке e_param_range_violation».

Кроме того, создание запроса помогает вам отделить важное от несущественного, благодаря чему ваш код зачастую становится лучше. Правда, он все еще ошибочный, но это уже куда более изящные ошибки.

Затем следует длинное описание, в котором вы излагаете свои гипотезы, предположения и идеи, которые хотите воплотить. Возможно, PHPschnargl вообще не особо популярное решение для вашей проблемы и весь остальной мир использует SozialGraf. Но ваши консервативные коллеги заставляют вас обращаться к PHPschnargl, потому что это стандарт, использующийся в компании уже много лет, потому что босс работал с этой библиотекой, когда еще был практикующим программистом, и потому, что никто не хочет разбираться с другой библиотекой. Это информация важна для каждого, кто станет помогать вам в решении проблемы. Если же вы ее утаите, то авторы первых ответов ограничатся тем, что просто просветят вас по поводу существования SozialGraf. Кроме того, этим вы лишаете читателей возможности предложить третий вариант, в котором в большей степени, чем в PHPschnargl, будут учтены ваши особые условия.

То же самое действует и для предположений, содержащихся в вашем подходе к проблеме. Если вы выразите их явно, это поможет окружающим найти в них ошибки и обратить на это ваше внимание.

Так что не пишите: «Когда я инициализирую PHPschnargl, то получаю сообщение об ошибке e_param_range_violation», а изложите проблему так: «Я инициализирую PHPschnargl с пустым объектом PersonalObjekt и массивом с классами, которые представлены URL. Я хочу получить заполненный PersonenObjekt и статус status_ok, но вместо этого получаю пустой PersonenalObjekt и сообщение об ошибке e_param_range_violation».

Расскажите обо всех шагах, которые успели предпринять для самостоятельного решения проблемы. Это не только поможет читателю понять ваши дей-

ствия и ход мыслей в поисках решения, но и заставит вас сделать эти шаги в принципе.

Иногда бывает так, что хоть в Интернете и можно найти много информации по проблеме, но на данном форуме решения тоже нет. Тогда вам следует скопировать адрес этой страницы и добавить его в просьбу о помощи в виде примечания: «Я уже читал это обсуждение, там эта проблема тоже описана, но решения нет». Этим вы не только показываете, что самостоятельно прорабатывали вопрос, но и исключаете возможность того, что вам посоветуют уже знакомые обсуждения.

В последнюю очередь укажите код, существенные ошибки и лог-файлы. Дайте как можно больше информации об операционной системе, версии языка и т. п. Некоторые ошибки общеизвестны и возникают только на одной-единственной платформе. Если в комьюнити вас открыто просят дать определенную информацию, обязательно выполните эту просьбу, даже если не совсем понимаете ее причин. Причина точно есть.

Важно предоставлять сокращенную версию кода. В идеале это должен быть самый маленький кусок кода, который еще вызывает описанную проблему. Новичкам зачастую бывает трудно сократить свой код, потому что они не делают резервных копий/бэкапов, не используют систему управления версиями и тестовую систему, с помощью которой можно проверить сокращенную версию кода, не меняя при этом бесповоротно оригинальную версию. Но то, что вы новичок, ни в коем случае не извиняет вас, наоборот, это должно стать стимулом улучшить свои методы работы таким образом, чтобы вы в любой момент могли протестировать код. Это значительно облегчит вашу профессиональную жизнь, потому что просто отпадет необходимость спрашивать совета на форумах.

Если вы разрабатываете веб-страницы или приложения, то разместите сокращенный пример кода на сайтах вроде jsfiddle.net или codepen.io. Там предлагаются три области для ввода данных: HTML, CSS и JavaScript. Если после проверки ваша проблема не была решена, то можете опубликовать ссылку на протестированный код. Преимущество этого сервиса в том, что здесь возможно все сразу: ваши читатели смогут без особых усилий просматривать код и последствия ошибки, а также отправлять вам ссылки на исправленный вариант.

Многие начинающие неплохие, а порой даже хорошие программисты испытывают чувство неловкости за свой код. Даже тех, кто не особо заинтересован в изяществе и эффективности своего кода, часто уязвляет его сравнение с примерами из учебников. Во время разработки программист испытывает тайное чувство стыда, но при этом он не обязан показывать свой код кому-либо еще. Но что делать, если у вас есть вопрос и вы хотите приложить к нему соответствующие части кода? Если у вас есть свободное время, можете попробовать причесать код, прежде чем отправлять его. Если же все нужно сделать быстро, то вам попросту придется смириться с тем, что весь остальной мир теперь знает о вашем маленьком грязном секрете. Лучше используйте свободное время для сокращения кода и утешайтесь тем, что многие из читателей не написали бы лучше, — в конце концов, примеры кода из учебников тоже выбраны не случайно, их писали люди с большим опытом специально под ситуацию в учебнике.

Перед тем как отправить вопрос, обязательно проверьте еще раз, вызывает ли вообще сокращенная версия вашего кода ошибку. Искушение пропустить этот шаг может быть велико, однако часто именно на этом этапе может сам собою появиться ответ на вопрос. Если это произойдет — на этапе сокращения кода или во время другой стадии подготовки вопроса — и если вы невероятно милый человек, то вы в любом случае сформулируете свой вопрос, дополните его решением и опубликуете.

Думайте о читателе

Итак, проблема вам известна и вы хотите быстро получить помощь. Однако разработчики, которые могли бы вам помочь, должны сначала вчитаться в ваше сообщение, поэтому снабдите свой вопрос информативным заголовком. Писать «ПОМОГИТЕ!!!» в заголовке неверно по разным причинам: во-первых, из-за прописных букв и избытка восклицательных знаков заголовок кажется навязчивым, во-вторых, он не дает никакой информации о содержании проблемы. А вот заголовок «PHPschnargl — проблема инициализации под Debian Shady» с бóльшей вероятностью позволит читателю заранее сказать, может ли он вообще здесь чем-то помочь. Не пишите в заголовке слова «Важно!» или «Срочно!», это приведет лишь к тому, что читатель почувствует себя принужденным и использованным. Срочность — это только ваша личная проблема, а не проблема тех, кто вам помогает. Также оставьте при себе предположения о том, что в вашей проблеме виноваты другие разработчики: «Эпический баг в PHP 5.0?»

Излагайте свои мысли кратко. Обычно мы вежливо предваряем сам вопрос какой-либо формальной вспомогательной фразой вроде «Э-эм, может мне кто-нибудь помочь, пожалуйста?» Благодаря этому у собеседника не возникает ощущения, будто на него вывалили всю информацию разом. Однако в случае форумов поддержки или чатов стоит переходить сразу к делу и опустить вводные вопросы, например: «Кто-нибудь здесь разбирается в?..» Ведь можно предположить, что на таких сайтах сидят люди, готовые помочь.

Старайтесь писать максимально грамотно. Включите проверку орфографии для используемого языка программирования. Запросы, к которым автор приложил хотя бы минимум видимых усилий, воспринимаются серьезнее, чем полная ошибок писанина.

Откажитесь от актов самоуничижения. Если вы выполнили свои домашние задания (см. ранее), то не нужно начинать свой вопрос с запевки: «Я знаю, что я глупый/глупая...», «Это точно моя вина...», «Я начал программировать только пять минут назад, но...». А если вы не подготовились, то пустые фразы не смогут убедить ваших читателей в обратном.

Когда вы прикрепляете данные, используйте распространенные открытые форматы. Не у всех есть лицензионный Microsoft Office, и не каждый сможет читать ваши файлы Word или Excel. Экспортируйте такие данные, например, в форматы HTML, PDF или CSV.

Не ожидайте слишком многого

Проще всего получить помощь там, где люди тусуются бесплатно. Вам не придется заключать договор на техобслуживание, составлять соглашение об уровне предоставления услуги и, конечно, что хорошо, не придется платить за услугу. Оборотная сторона медали в том, что все эти люди на форумах жертвуют свое время на добрые дела. И если вы зададите вопрос в духе «Может мне кто-нибудь объяснить принцип действия параметра X?», то тем самым вы попросите ближних своих потратить очень много времени и написать довольно длинное сообщение. Это срабатывает редко и является неуважением к другим людям.

Гораздо лучше вопрос вида «Может кто-нибудь дать ссылку на объяснение работы параметра X, которое будет понятно новичку?». А еще лучше, конечно, самому поискать эту ссылку.

Если вы не сразу поняли ответ, то попробуйте сначала помочь себе сами: поищите и хорошенько его обдумайте, прежде чем переспрашивать у автора. Как только вы нашли человека, готового помочь, появляется большое искушение не раздумывая обращаться к нему и с другими вопросами.

Пожалуйста, не откусывайте никому руку по локоть только потому, что вам протянули мизинец помощи. Для дальнейших вопросов действует все тот же принцип вежливости — принцип самостоятельной предварительной подготовки. Чем точнее вы сформулируете просьбу о помощи, тем больше шанс, что вы ее получите. Хорошие вопросы — это вопросы об инструкциях (которые трудно найти), о рабочих примерах кода или о том, не хочет ли кто-нибудь глянуть на небольшую, в идеальном случае даже интересную, проблему и помочь в ее решении.

Ни в коем случае не ждите, что вам отладят код и предложат абсолютно готовое решение для приблизительно описанной проблемы. На форумах по программированию ненавидят запросы типа «Отправьте мне код, пожалуйста», потому что авторы таких постов бессовестно пытаются свалить свою работу на других. Любое интернет-сообщество может стерпеть такие сообщения, но в долгосрочной перспективе слишком большая терпимость приводит к тому, что хорошие люди уходят из этого сообщества и оно распадается.

Не приводите абстрактных примеров

Часто авторы описывают свои *предположения* о том, в чем может заключаться проблема, и при этом пренебрегают самим *описанием* проблемы. Это не особо эффективно по двум причинам. Во-первых, лучше бы они использовали время и место, потраченные на свои предположения, на реальную информацию, максимально свободную от домыслов.

Из-за того что авторы путают предположение о проблеме и описание проблемы, у них возникает приятное чувство, что вопрос задан достаточно ясно. Но на самом деле они представили только свою точку зрения на положение вещей. Во-вторых, такое описание проблемы направит читателя по ложному следу. Даже если предположение спрашивающего не имеет ничего общего с реальной проблемой, оно

сразу закрепится в сознании читателя. И поначалу он будет искать решение именно в этом описанном направлении. А поскольку ваше предположение оказалось ошибочным, то высока вероятность того, что и отзывчивый читатель тоже не сможет найти реальную причину.

Помогающему будет гораздо легче увидеть истинную проблему не через призму чужих предположений о проблеме.

Оставайтесь вежливы несмотря ни на что

Не всегда ответ дается в том ключе, в котором хотелось бы нам. Причины этому могут быть разные, например, вопрос задан неточно, или не в том месте, или на него уже отвечали 9000 раз. Тогда кратко извинитесь и в случае необходимости предоставьте недостающую информацию. Если отзывы разные, то реагируйте только на конструктивные и игнорируйте грубые и явно глупые ответы. Но будьте осторожны: то, что на первый взгляд кажется грубым и откровенно бессмысленным, может оказаться правильным, но просто неприятным для вас ответом. Попробуйте разобраться, у вас плохое настроение, потому что ответивший обидел вас или потому что он прав, а вы — нет.

Если вы получаете только отрицательные или грубые ответы, но при этом не видите какой-либо своей ошибки или если не можете понять, почему обвинения против вас могут оказаться справедливыми, возможно, у всех ответивших сегодня плохой день или они сами по себе грубияны. Даже у интернет-сообществ есть свои хорошие и плохие периоды, а в конце их существования от изначального желания помогать там почти ничего не остается. Избегайте гневных или разгромных ответов со своей стороны, какими бы заманчивыми они ни казались в первые минуты негодования. Избегайте их даже тогда, когда у вас на языке вертится самое остроумное замечание в мире. Мы знаем, как тяжело это сделать, и сочувствуем вам. Однако зачастую, в том числе в Интернете, самым мудрым шагом будет просто не обращать внимания и тем самым погасить назревающую ссору — любой ответ только раздует пламя еще больше, а не приблизит вас к вашей цели, вы только потратите свои время и нервы. Дополнительное преимущество этой стратегии — после нескольких десятилетий практики вы достигнете духовного состояния буддийских мудрецов, освободитесь от цикла страданий живых существ и сможете встать на профессиональный путь просветленного мастера[1].

Если вам в голову приходят лишь глупые ответы или их вообще нет, воздержитесь от того, чтобы во второй раз задавать тот же самый вопрос, только приправленный упреками. Попробуйте задать его в другом месте. Если и там история повторится, высока вероятность того, что проблема все-таки в вас. Тогда прочитайте эту главу еще раз с самого начала.

Если вы получили конструктивные ответы, поблагодарите тех, кто вам помог, и напишите краткие выводы, которые могут помочь в будущем другим программи-

---

[1] Тогда вы автоматически получите ответ на все свои программистские вопросы: «Проблемы не существует».

стам, у которых возникнет тот же вопрос. Так вы сделаете мир лучше, возвращая частичку того, из чего извлекли пользу сами.

Выводы должны идти под таким же заголовком, что и исходный вопрос, только с припиской «РЕШЕНО» или SOLVED. Те, у кого мало времени, смогут пропустить это сообщение, если заранее увидят, что в нем только выводы и благодарность. А те, кто не заинтересован в обсуждении решения, сумеют сэкономить много времени, прочитав только исходный вопрос и решение. Даже если поисковики не выдали результатов, а на соответствующих форумах поддержки никто не смог или не захотел помочь и пришлось самостоятельно решать проблему в поте лица своего, то именно тогда вы должны сделать над собой усилие. Еще раз доступно опишите путь решения проблемы и увековечьте его в подходящем месте в Сети. Этим местом запросто может быть и ваш собственный блог, если проблема достаточно специфична для того, чтобы в будущем у пользователей был шанс быть перенаправленными со страницы поиска сразу к вам. Мир отблагодарит вас за это (иногда даже в виде восторженных писем).

9 Право на оказание помощи

За каждым «Понятно» скрывается маленькое «Чего-о?».

Matthias Rampke/@matthiasr, Twitter, 21 февраля 2011 года

Даже самая непросвещенная личинка программиста знает кого-нибудь, кто знает еще меньше. Поэтому в сообщениях вроде «Сейчас я тебе все быстренько объясню» плохим программистом может оказаться не только тот, кому требуется помощь, но и тот, кто объясняет. В некотором смысле это преимущество, потому что, как у плохого программиста, у вас есть редкая возможность, если нужно, поставить себя на место человека, которому требуется какое-либо объяснение. Продвинутые программисты зачастую уже не помнят, каково это — с остекленевшим взглядом вслушиваться в непонятные объяснения. В этой главе речь пойдет о том, почему благое намерение помочь так часто оказывается неудачным и как лучше поступить, если вы сами окажетесь на стороне объясняющих. Вы также спокойно можете дать почитать эту главу штатным программистам, которые обычно охотно вам помогают. Может быть, это облегчит достижение взаимопонимания между вами.

Неправильная причина

В самом безобидном варианте неудавшейся попытки объяснения более хороший программист хочет помочь собеседнику, но у него не выходит. Знать тему — еще не значит уметь объяснять ее. И после провалившейся попытки объяснения собеседник оказывается в еще худшей ситуации, чем раньше. Перед разговором у него, возможно, вообще не было представления о *концепции MVC*, а после разговора он следующие пять лет будет повторять: «Ох, я когда-то уже пытался это понять, но это было слишком сложным для меня». Таким образом, лучше сначала подумать, действительно ли вам следует пытаться что-либо объяснять другому человеку.

КОНЦЕПЦИЯ MVC

MVC, аббревиатура от Model/View/Controller (модель/представление/контроллер), — это устоявшийся в сфере профессиональной разработки программного обеспечения шаблон проектирования. Концепция MVC предполагает следующее. Данные хранятся структурированно согласно некоторой модели (Model), представление (View) отвечает за отображе-

ние данных, а контроллер (Controller) — за то, как программа реагирует на запросы пользователя. MVC используется в программах с графическим интерфейсом для того, чтобы аккуратно разделить код, относящийся к различным задачам. Подробнее: en.wikipedia.org/wiki/Model-view-controller~~pobj.

В особенности это касается объяснений, о которых вообще никто не просил. Ведь тема не становится априори интересной только потому, что вы сами считаете ее таковой. И если вы в восторге от лямбда-исчисления, то ваш энтузиазм не передастся остальным, если вы просто будете долго убеждать их в этом. Интерес — это такой застенчивый зверь, который приходит к людям добровольно или не приходит вообще. И абсолютно нормально, что кому-то может быть плевать на тему, в которой вы ас, даже если за этой темой будущее. У этого человека, возможно, мало времени, или достаточно собственных интересов, или люди из его окружения тоже частенько пытаются привлечь его внимание к своим любимым темам. Поэтому вы должны исходить из уровня человека и учитывать его (предполагаемые) интересы. При этом нельзя забывать одного: и у наших собственных увлечений нет рациональных причин, хоть какие-то мы потом себе и додумываем. Определенно, у вас есть хорошие основания считать, что каждый человек должен разбираться в архитектуре процессора, если он хочет найти свое место в XXI веке. Но, возможно, вы даже не осознаете, что ваше увлечение этой темой вызвано чем-то настолько несерьезным, как, например, детские воспоминания о шоколадках в форме процессора. И есть множество других полезных для жизни в XXI веке знаний, которых у вас нет.

Тот, кто хочет чему-либо научить других против их воли, должен был становиться учителем. А общаясь с равными себе взрослыми людьми, не остается ничего другого, как смириться с отсутствием у собеседника интереса к вашим выдающимся особым знаниям. В противном случае желание объяснять — это чистое тщеславие, а тщеславие — это всегда некрасиво. Если человек — чрезвычайно умелый наставник или собеседник так или иначе боготворит его, то в редких случаях удается зажечь искру энтузиазма. Хотя это скорее исключение из правил.

Вот как это работает: помогаем, даже если не просят

Особый случай представляют собой проблемы, которые человек считает неразрешимыми и потому стоически их терпит. Сам он о помощи не попросит. Но если все умело обставить, то такую проблему можно распознать и решить. Как это сделать, Йоханнес объяснит на одном примере.

«После того как мои предыдущие попытки перевести маму с ОС Windows на Mac OS (что делалось под влиянием технологического евангелизма) потерпели крах из-за отсутствия конкретных наглядных преимуществ Mac OS, к переходу с Internet Explorer на Firefox я решил подойти иначе. Сперва я просмотрел, какие сайты мама использует постоянно. Портал T-Online был одним из этих сайтов и в особенности подходил для моей цели, потому что представлял собой перегруженный флешами юзабилити-ад. Моя мать к тому же легко отвлекается, поэтому мельтешащие флеш-баннеры вызывают у нее

большие проблемы с концентрацией. Так что я в общих чертах объяснил ей преимущества Firefox, после чего, заверив ее, что ничто кардинально не изменится, и пообещав все начисто удалить, если ей не понравится, установил Firefox с расширением для блокировки рекламы Adblock Plus.

Потом я открыл ей портал T-Online сначала в Firefox, а для сравнения — в Internet Explorer. После чего только и нужно было, что перенести закладки и вкратце объяснить маме, где в Firefox находятся кнопки. Затем я быстро удалил Internet Explorer с панели быстрого запуска, добавил на его место Firefox и еще недели спустя радовался маминому счастью».

Чем этот частный случай отличается от нежелательного и непрошеного вмешательства в жизнь других людей? Во-первых, Йоханнес интуитивно понял, в чем заключается проблема, которая была у его матери. И не он один считал, что это действительно проблема. Во-вторых, ему удалось продемонстрировать конкретное преимущество. И в-третьих, он оставил за ней выбор между старым и новым, а не поставил ее перед свершившимся фактом.

Корыстные мотивы

Если на вопрос «*Можно* ли помогать?» мы ответили, то пора переходить к вопросу *«Зачем?»*, поскольку побуждения человека, который что-то знает и хочет эти знания передать, редко бывают абсолютно благородными и чистыми[1]. В целом не одобряется, если кто-то, помогая другому, помогает самому себе и ожидает за это благодарности. Но именно это часто происходит во время объяснений и обучений, в особенности что касается технической сферы.

Могут существовать следующие корыстные мотивы.

- Вы хотите привлечь на свою сторону больше новых людей («Просто всегда используй язык программирования Ruby!»), при этом осознанно не обращая внимания на то, что другой подход к решению их проблемы подошел бы, возможно, гораздо больше. Для фанатов молотка каждая проблема выглядит как гвоздь, во всяком случае, до тех пор, пока через несколько недель они не обнаружат, что гаечный ключ еще круче.
- Инвестиции личного времени должны быть оправданы в последующем. Вы провели месяцы, углубленно изучая тему NoSQL, хотя — ну ладно, потому что — вам вообще-то вместо этого требовалось срочно завершить диссертацию. Все это не должно оказаться напрасным! Остальные должны убедиться в важности NoSQL!

---

[1] Авторы этой книги, например, не так уж и заинтересованы в дальнейшем развитии своих читателей. Они хотели из личного тщеславия опубликовать книгу в издательстве O'Reilly, по возможности с каким-нибудь животным на обложке, ослом например. И в том случае, если бы им в руки попала машина времени, они хотели бы передать самим себе в прошлом полезную книгу.

- На самом деле вы вообще не хотите, чтобы другой человек чему-нибудь научился. Просто хотите продемонстрировать, что вы умнее его. И хотите, чтобы так оно и оставалось. Так оно и останется, потому что подобным отношением вы не только отпугиваете большинство собеседников, но и неосознанно противодействуете процессу передачи знаний. Например, если вы, помимо всего прочего, изображаете проблему более сложной, чем на самом деле. Потому что это единственный способ представить в действительно выгодном свете тот факт, что вы самостоятельно разобрались в этом сложном вопросе. Если в достаточной степени драматизировать сложность проблемы, то ее решение, возможно, даже будет денежно вознаграждено.

Конечно, у каждого нашего поступка есть множество причин. В итоге всегда приходится иметь дело с мешаниной из разных побуждений, славных и не очень. Граница между альтруистической и эгоистической мотивацией особенно тонка, когда речь идет об извечной потребности в сексе. Помогая, вы показываете себя с лучшей стороны — вы дружелюбны, остроумны и терпеливы. Что ж, неплохие предпосылки для обучения. В то же время вы хотите не столько помочь своему собеседнику как можно лучше разобраться с системой TYPO3, сколько заполучить его в свою постель. И разве его восхищение вашими обширными знаниями не стало бы подспорьем в этом деле? Если бы он стал зависеть от вас в чем-то одном, не помогло бы это в итоге продлить ваши интимные отношения? А если все в итоге получилось, тогда: «Малыш, тебе больше не нужно изучать TYPO3. Теперь у тебя есть я. Я все сделаю».

Отсутствие эмпатии

Вас и впрямь попросили помочь. Вы подумали о собственных мотивах и нашли их вполне порядочными. Что дальше? Сперва потратьте немного времени на то, чтобы узнать, чего именно хочет просящий вас о помощи человек, что ему нужно. Может быть, это нечто совершенно не то, чего хотели бы вы сами, окажись вы на его месте. Предположим, вы в восторге от любой новой технологии и мысль, что ваши друзья могут выбрать не самое лучшее техническое решение проблемы, невыносима для вас. Ваш собеседник, напротив, может быть одним из тех, кто укоренился в своих привычках и совсем не любит пробовать новое. Технологии для него лишь средство достижения цели. Обычно он не проявляет интереса даже к каждой отдельной детали под капотом. Невыносимая ситуация! Вы обязаны растолковать бедняге, насколько ничтожным было его предыдущее решение и как срочно ему требуются новый компьютер, новая операционная система и абсолютно новый язык программирования. Только так он сможет из головастика стать настоящей лягушкой-программистом. Если разговор перешел в эту фазу, то тот, кому нужна помощь, как правило, вырубается. Ведь он не хочет ни давать себя в обиду, ни начинать новую жизнь. Он просто ищет понятное для себя решение видимой проблемы. Если вы не готовы придумывать это решение вместе с ним и пойти на риск того, что оно может включать такие шокирующие вещи, как создание HTML-страницы в Word, тогда сделайте вам обоим одолжение и завершите беседу. Вы ведь можете остаться друзьями.

Если не сломано — не трогай

Один из авторов этой книги проработал несколько лет в отделе концерна. В этом отделе, оборота которого вполне хватило бы на довольно беспечную жизнь, все рабочие процессы были организованы с помощью непроходимого леса из Excel-таблиц, VBA-макросов и состряпанных практикантами Ruby-сценариев. Серверов, SQL и эффективной архитектуры просто не было. Тем не менее этот отдел справлялся со своими обязанностями, хоть зачастую и неэффективно, однако достаточно для того, чтобы его не постигла судьба плановой экономики СССР. Интересно, что сотрудники часто находили довольно креативные способы, как обойти ограничения и проблемы с эффективностью их Workflow.

Если смотреть извне, ситуация во многих отношениях была невыносимой, но некоторые сотрудники обзавелись удивительными способностями, позволившими им сделать из MS Office инструмент для анализа данных. Попытка заинтересовать руководителей проекта и распорядителей бюджета инструментом для анализа, который был бы тщательно разработан с нуля, провалилась с треском. Но причиной этому были не проблемы с бюджетом или отсутствие интереса к оптимальным структурам, а нежелание менять частично работающую систему на неясные обещания.

Вот что, помимо глубоких знаний Excel, автор вынес из этого опыта прежде всего: идеальное решение не всегда подходящее. И тот, кто дает совет, должен уметь иногда сдерживаться и вносить улучшения постепенно и понемногу, начиная с перил лестницы, а не сносить сразу все здание. В данном конкретном случае, конечно, помогло еще и то, что у концерна были деньги на покупку дорогостоящих рабочих станций. Это помогло сократить вычисления с 6 часов до 1,5 часа, и в это время можно было спокойно пообедать.

Слишком много информации сразу

Когда вы выяснили, что на самом деле требуется вашему собеседнику, подумайте о том, какие знания вы хотите дать, а затем мысленно вычеркните две трети этого списка. Человеческий мозг не способен сразу справиться с неограниченно большим количеством новой информации и свежих идей. И тот, кто объясняет другому человеку два-три небольших вопроса в день, уже сделал многое. Ограниченность человеческого восприятия — щекотливая тема для обеих сторон. Тому, кто выступает в роли наставника, будет неприятен собственный провал, когда уже после введения слушателю потребуется многодневная пауза для обдумывания. А тот, кто хочет что-то узнать, почти всегда считает, что другие поняли бы за это время в десять раз больше, но сам он немного несообразительный. В обязанности наставника входит определение уровня возможностей человека. Необходимо составить скромный план («сначала объяснить, что такое функция»), а затем противостоять искушению пойти гораздо дальше. То же самое верно и для скорости объяснения: оцените, сколько времени у вас уйдет на объяснение какого-то одного вопроса. Удвойте это время. Если вы закончили объяснять до того, как оно истекло, — вы перегрузили свою аудиторию. (Если вы попали в ситуацию, когда приходится говорить перед группой людей, которых

работодатель или другой авторитетный человек принудил слушать вас, то удвоенного времени недостаточно. В таком случае можете смело увеличить время запланированного выступления в десять раз.)

На этом месте остановитесь еще раз. Вы по-прежнему отталкиваетесь от своего уровня знаний, на который вы хотите поднять собеседника, а не от его собственного уровня. Путь наверх тернист, поэтому с легким сердцем вычеркните еще несколько аспектов. Не поддавайтесь искушению начать объяснение с абстрактных принципов. Конкретные примеры — вот ваши друзья. Для абстрактных понятий всегда найдется время в конце. Выбирайте эти примеры таким образом, чтобы как можно меньше запутывать аудиторию не относящимся к делу интеллектуальным балластом. Упрощенные модели а-ля Hello, World щадят восприятие.

На каждом этапе своего объяснения проверяйте, понимает ли собеседник, что вы имеете в виду. И не думайте, что, если он кивает, значит, он уже во всем разобрался. Возможно, он просто не хочет признавать, что понятия не имеет, что вы подразумеваете под массивом. Поскольку вы сами впервые заинтересовались этой темой в возрасте семи лет, то теперь не помните ни о том, каково это — не иметь ни малейшего понятия ни о массиве, ни о том, сколько времени вам потребовалось, чтобы действительно понять эту концепцию. А уточнения «Ты знаешь, что такое массив?» недостаточно. Сотрудники кол-центра не спрашивают: «А вилка воткнута в розетку?» — потому что это обидит любого звонящего и он, не проверяя, ответит: «Конечно, в розетке, я же не идиот». Если спросить вместо этого: «Вы не могли бы проверить, какой формы разъем?» — то собеседник заползет под стол и обнаружит, что вилка вообще не вставлена в розетку. Поэтому формулируйте уточняющие вопросы тактично и так, чтобы собеседник не смог рефлекторно ответить: «Я знаю».

Отвечайте на конкретные вопросы

Предположим, вам задали вопрос: «Какое снаряжение мне необходимо, чтобы подняться на восьмитысячник?» Сейчас у вас есть два варианта: можете дать подробный технически правильный ответ: «Год тренировок, противоперегрузочный костюм, кислород» — или впасть в недоумение из-за высоты горы. Не выдумывайте причину, почему спросившая вас хочет забраться именно на восьмитысячник, спросите ее сами. Так вы дадите ей шанс поведать, что она просто хотела приготовить чашечку чая и высчитала, что на высоте 8000 метров вода кипит при комнатной температуре. И тогда вам понадобится всего пара секунд на ответ: «Легче всего это сделать с помощью кипятильника». Однако обдумывание ответа не всегда такое простое, как в нашем примере. Именно не очень хорошие программисты часто получают на свои вопросы ответ, что их подход к проблеме в корне неверный («Ты делаешь это неправильно!»). Хотя, возможно, у них была конкретная причина выбрать именно этот подход. Причина в вопросе может быть не указана. А если и указана, то вам она, возможно, кажется очень плохой. Но в ситуации, в которой находится собеседник и о которой вы, вероятно, знаете недостаточно для того, чтобы сформировать свое мнение, отвратительный подход к проблеме тем не менее может оказаться подходящим.

Кроме того, вы можете обидеть собеседника, если, словно работающий с текстовыми шаблонами сотрудник техподдержки первого уровня, будете сразу исходить из того, что он не понимает, что делает. Компромиссом здесь было бы сказать, почему подход собеседника нельзя считать лучшим вариантом, и объяснить решение, которое представляется вам более элегантным. Вежливо сформулируйте обе эти части, не пишите: «Ваше решение — это решение для идиотов, а вот так это делаем мы, настоящие программисты». Если собеседник кажется вам невежливым, агрессивным или глупым сверх всякой меры, не злитесь. Ведь именно тогда, когда речь заходит об особенно острых проблемах (пролили кока-колу на клавиатуру, упал жесткий диск, в товарной базе данных пропали входящие заказы), стресс может отключить на время способность думать и/или хорошие манеры даже у обычно приятных людей. И даже если вам очевидно, что спрашивающий — просто чертовски ленивое создание, которое, перед тем как спросить, элементарно не задало свой вопрос в поисковике, не отвечайте только: «RTFM». Дайте ссылку или по меньшей мере правильные ключевые слова для поиска, причем это должна быть не насмешливая ссылка на lmgtfy.com. Если речь идет о разговоре вживую, в котором вы не можете просто отмолчаться, как при переписке, то разумным выходом станет дружелюбно сказанное: «Это ты легко можешь выяснить и сам». Так вы завершите раздражающий разговор и не будете чувствовать, что вас использовали. Если вы предполагаете, что ваш ответ на глупый или заданный в неправильном месте вопрос может унизить собеседника, не отвечайте при всех. Отправьте ему сообщение. Если он присутствует лично, отвечайте только тогда, когда никто больше вас двоих не слышит. Мы могли бы сейчас привести аргумент, что эффект от обучения выше, если обучаемого не подвергать публичному осуждению, но вообще-то вам должно хватить и того, что этого требует нормальное человеческое вежливое отношение. Вспомните, как много было разных ситуаций (в автомастерской, фотолаборатории, на праздничном собрании лесников), когда вы сами просто по незнанию вели себя неправильно. Если в своем вопросе автор пишет, что уже пытался найти ответ, но безрезультатно, не считайте, что он искал его с ведром на голове. Вы и все, кто работает в свободном проекте GNU sperflocken, знаете, где точно располагаются документы, FAQ или ответ на вопрос. Но очевидно, что незнающему человеку не так-то просто найти эту информацию. Что можно изменить, чтобы облегчить поиск для следующего человека? Может оказаться и так, что на этот вопрос пока еще никто не ответил, но вы можете сделать мир лучше для людей с такой же проблемой (параллельно избавив себя и других от обязанности объяснять то же самое в будущем). Дополните документы или отправьте сообщение ответственным за этот вопрос лицам, в котором тщательно сформулируйте предложение по улучшению ситуации.

Если вы сами не знаете ответа

Помощь — вещь хорошая. Настолько хорошая, что мы часто не удерживаемся от соблазна помочь, даже если сами совсем не знаем ответа, и начинаем просто гадать. В случае, если грамотных ответов еще нет (и если вопрос не был задан письменно десять минут назад или устно две секунды назад), вы можете помочь собеседнику и догадкой тоже — при условии, что вы достаточно честны, чтобы уточнить в своем

ответе, что речь идет именно о предположении. Если вы самостоятельно искали решение и нашли его где-либо еще, то не делайте вид, будто появились на свет с этим знанием. Укажите путь решения проблемы, возможно, это научит собеседника большему, чем ответ сам по себе. Если вы не знаете ответа и не хотите его искать, тогда, пожалуйста, не стойте на пути спасателей. В особенности воздержитесь от шуточек, которые наивный собеседник может принять за ответ. В качестве плохого примера можно привести то воодушевление, с которым участники мотофорумов автоматически реагируют на вопрос «В моем мотоцикле скрипят тормоза. Что делать?»:

- Constantin H., 30 апреля 1998, 9:00: «Масло помогает в том числе. Удачи!»;
- Kungfufighter, 1 января 2008, 20:53: «Самое простое решение — снять всю обшивку и немного смазать маслом»;
- Cheffee, 8 февраля 2011, 14:57: «Масло необычайно снижает скрип».

Восхитительная шутка, и вот уже лет десять как находится кто-нибудь, кто считает ее вершиной среднеевропейского ситуативного юмора. На самом деле это не только слабенькая, но и потенциально опасная шутка.

Если ваши коллеги худшие программисты, чем вы

Если однажды вы станете неплохим или даже хорошим программистом, то, возможно, попадете в ситуацию, когда вам придется работать с менее сильными коллегами. Умение воспитанно вести себя с этими людьми и, может быть, даже помогать им стать лучше встречается настолько редко, что, если у вас получится, это сделает вас необычайно популярным. Ваша слава будет далеко опережать вас, а потомки станут вспоминать вас с благодарностью. Вам просто необходимо учесть следующие советы.

Достаточно хорошо уже достаточно хорошо

Психоаналитик Дональд Винникотт в 1950 году ввел понятие «достаточно хорошей матери». Его суть заключается в следующем: пока ребенок еще беспомощен, мать удовлетворяет каждую его потребность. С развитием у ребенка навыков мать больше не реагирует так быстро на его желания, и таким образом ребенок постепенно становится самостоятельным. Если перенести это на ситуацию общения с начинающими программистами, то смысл заключается в том, что с какого-то момента больше не надо реагировать на каждый призыв о помощи. Это получится с легкостью, если вы не будете задерживаться на одном месте или ответите на e-mail или сообщение с просьбой о помощи не сразу, а спустя некоторое время. Но даже если автор вопроса сидит за соседним столом, вы можете попросить его подождать минут 15, пока вы заняты чем-то другим. За эти 15 минут он, возможно, сам найдет ответ на свой вопрос.

«Смотри, это совсем просто»

О следующей проблеме вы, наверное, знаете, из собственных наблюдений в других сферах вашей жизни. Повара-любители рассказывают о том, как легко самим приготовить макаронной суп, начиная с бульона и лапши собственного приготовления

и заканчивая травами, выращенными на балконе. Конечно, не так уж сложно приготовить суп с лапшой, но перед тем, как перейти непосредственно к готовке, нужно сделать много шагов, о которых профессиональный повар уже просто не помнит: покупка специальных кухонных приборов и ингредиентов, которые «конечно, есть в каждом доме». Важным и, возможно, самым трудным условием является формирование такого психического состояния, в котором человеку в принципе захотелось бы самому приготовить суп с лапшой, вместо того чтобы залить кипятком содержимое упаковки. А фразу «Это же совсем просто» можно перевести так: «Я совершенно забыл, какие условия мне пришлось создать и что я должен был дополнительно изучить, прежде чем эти действия стали для меня сами собой разумеющимися»[1].

Кроме прочего, вы должны сподвигнуть своего «ученика» на то, чтобы он начал задавать вопросы самостоятельно и по возможности не казался при этом самому себе глупым. Как правило, для этого недостаточно просто убедить собеседника в том, что глупых вопросов не существует, ведь, если честно, все знают, что это неправда, особенно если вы время от времени сплетничаете о плохих образцах кода. Если же вы, напротив, сделаете вопросы и ответы естественной частью сотрудничества, дело пойдет куда лучше. Особенно хорошо, если вы будете доброжелательно реагировать на робкие вопросы («Очень хороший вопрос...») собеседника и после того, как ответите на них («Это был очень хороший вопрос, потому что...»), поделитесь с ним некоторыми базовыми знаниями, которые в последующем облегчат ему жизнь. И таким образом он не только узнает парочку небольших фактов по своему вопросу, но и ознакомится с более широким контекстом.

Плохой код? Сохраняйте спокойствие

Не стоит с ходу полностью переписывать код за худшим, чем вы, программистом. Ни в присутствии автора, ни в его отсутствие. Большинство воспримет это не как помощь, а как оскорбление. Программист, ошибку которого вы так масштабно исправили, вряд ли захочет взять себе на заметку что-нибудь из изящно переписанного вами кода, скорее он подумает о переводе в отдел маркетинга.

Если же код просто ужасен, но отсутствие качества никак не повлияет на весь проект, купите себе резиновый мячик для снятия стресса и молчите. Если вы работаете над одним кодом, то подайте хороший пример своей работой. Благодаря этому ваш коллега, не спрашивая, сможет узнать что-то новое, не потеряв при этом

---

[1] И наоборот, это также означает, что все на самом деле просто, если вы готовы сделать множество маленьких шагов. Джо Армстронг, создатель языка программирования Erlang, рассказывает: «Некоторые говорят: “Компилятор — это же так трудно”. Совсем нет. Это легко. Есть масса мелочей, которые нетрудны и которые нужно освоить. Надо разбираться в структурах данных. Надо разбираться в хеш-таблицах и в парсинге. В генерировании кода. В техниках интерпретации. Каждый из этих предметов не особенно сложен. Начинающие думают, что это все большие и сложные темы, и поэтому к ним не подступают. Все, что вы не делаете, трудно, все, что вы уже сделали, легко. Люди даже не пытаются ничего делать, и, я думаю, это ошибка». (*Сейбел П.* Кодеры за работой. — М.: Символ-Плюс, 2011. — С. 206)

лицо. Полезным будет также вдвоем обсудить код с третьим, в идеале незнакомым с вами лично человеком (см. раздел «Коллективное чтение кода» главы 7). Вы можете рассказать, как представляете себе качественный код, не оскорбляя при этом присутствующих. (Не говорите, что чужой код явно писало стадо пробежавших по клавиатуре диких кабанов. Собеседникам, обладающим хоть малейшей социальной компетентностью, будет понятно, что за их спинами вы точно так же говорите и об их кодах.) Но не сваливайте на своих бедных коллег все стандарты программирования за раз. Одной-двух идей в день хватит. Никто не сможет переварить больше.

Если вы вместе с коллегой сидите перед одним компьютером, потому что практикуете парное программирование или рефакторинг (см. главу 15), не исправляйте его сразу же, как только он выбрал не самое оптимальное решение или допустил ошибку. Но и не заставляйте себя полностью отказаться от замечаний, если ваш напарник боязливый или скромный человек, который перед каждой строкой говорит: «Мимими, вот это точно неправильно!» — и вопрошающе смотрит на вас. Обратите внимание на проблему и подождите. Если однотипные проблемы будут повторяться, то, возможно, даже получится установить их причину. Эффект от обучения будет больше, если напарник сам заметит, что требуется улучшить. И самое важное: представьте себе, что клавиатура — это кошка, которая пару недель назад попала под машину. Вы должны любой ценой справиться с желанием взять ее на руки, сядьте на них, если придется.

Как помешать успешному обучению

Самые эффективные методы, как удержать людей от обучения.

1. Отпилить голову ученика бензопилой.
2. Отобрать у ученика клавиатуру.
3. Сделать языком преподавания клингонский (только не у клингонов).

Самый универсальный совет в мире гласит: не тяните одеяло на себя! В переносном смысле это правило действует и тогда, когда вы хотите научить кого-либо, как правильно чинить велосипед, паять или пеленать ребенка. Действует оно и для клавиатуры — неправильно тянуть ее на себя.

И если вы из всей главы запомните только этот один-единственный пункт, мы уже будем счастливы.

10 Как выжить в команде

> Собрания с самим собой проходят по большей части без проблем. Тем не менее придумал парочку запасных вариантов.
>
> *@roarrrbert_we/Twitter, 16 ноября 2011 года*

В жизни каждого неплохого программиста однажды наступает день, когда он впервые начинает работать с другими программистами. Это непростой день. Для многих это вообще первый раз, когда другой человек увидит их код и выскажет свое мнение. Это может стать первой встречей с системой управления версиями или первым опытом общения в гетерогенной среде («О боже, у остальных Mac!/Linux!/Windows!»). Человек, который хочет, чтобы в коллективе его признали ценным сотрудником, сталкивается с недоброжелательностью и впервые знакомится с различными подходами к работе. Собственные методы, самостоятельно разрабатывавшиеся годами, и произвольно сформировавшийся стиль программирования впервые подвергаются критической оценке. Даже если вы в совершенстве владеете своим языком программирования — а это совсем не так, иначе вы бы читали другую книгу, — в совместной непривычной для вас работе в команде таится множество ловушек.

На этом месте напрашивается возражение: «Тот, кто читает эту книгу, в любом случае не работает в профессиональной команде разработчиков программного обеспечения». Но это далеко не так! Особенно в новых сферах на работу часто от безысходности нанимают всех, кто способен хотя бы произнести по буквам название языка программирования. Может случиться и так, что вам сделают предложение, от которого вы не сможете отказаться.

> «Все же нам нужны еще несколько человек, которые смогут в понедельник где-то в земле Мекленбург — Передняя Померания набрать на клавиатуре Java-подобные буквосочетания». Ответы от трудовых бирж на этот запрос становились все смехотворней. На рынке профессиональных программистов, казалось, разобрали все кадры. Когда мы ничего не добились и с помощью рассылки электронных писем широкому кругу знакомых своих сотрудников, мне в голову пришла идея пойти с пачкой денег на отделение информатики в одном университете, чтобы на месте нанять студентов на работу. [...] «Вот. Десять тысяч. Это начальный бонус для тех, кто в понедельник поедет в Засниц и будет там программировать на Java. И подпишет полугодовой контракт, конечно».
>
> *Саша Лобо. Вспышка*

Между тем разработка программного обеспечения стала частью многих профессий, к которым она не имела абсолютно никакого отношения еще пару лет назад. Можно встретить аспиранта, который должен создать для своей диссертации небольшой инструмент, или графического дизайнера, который должен уже не просто сдать проект, сделанный в Photoshop, но и уметь конвертировать его в HTML и JavaScript. Если вы работаете вместе с Back-end-разработчиком, который отвечает за серверную часть сайта, который у вас заказал клиент, то тогда вы внезапно становитесь частью команды разработчиков. От ученых-естественников часто ждут, что они создадут собственные инструменты для программного обеспечения. Это звучит безобидно, но может иметь неприятные последствия.

> С такой проблемой столкнулась одна из рабочих групп научно-исследовательского института Скриппса (Ла-Хойя, Калифорния). Команда под руководством Джеффри Чанга изучала химические структуры. В 2006 году они обнаружили признаки ошибки в программе, которую разработали сотрудники другой лаборатории. Ошибка привела к тому, что в двух столбцах входные данные получали неправильный знак, и из-за этого якобы происходила инверсия кристаллических структур молекул белка, а исследователи стремились именно к этому. [...] Команде Чанга пришлось опровергнуть полдесятка своих заявлений в изданиях Science, Journal of Molecular Biology и Proceedings of the National Academy of Sciences. С тех пор они все проверяют и перепроверяют по нескольку раз.
>
> *Computational science: ...Error — Why scientific computing does not compute» / www.nature.com/news/2010/101013/full/467775a.html*

Даже если вы когда-то уже работали в команде, а затем перешли в новую компанию, все равно может появиться чувство неуверенности. Здесь действуют те же правила, что и в старой компании? Я покажусь всезнайкой, если предложу правила, принятые в предыдущей рабочей группе, или на меня косо посмотрят, потому что они давно устарели?

Если вы начинающий разработчик, то лучше проведите первое время в новой команде за чтением и учебой. Читайте всё: исходный код, документацию, должностные инструкции, организационную структуру и билд-скрипты. Учитесь у других, как можно чаще подсаживайтесь к коллегам — тем, кто пообщительней, — и следите за каждым их движением.

Во многих командах это нормальная практика, но далеко не во всех. Если нет, то стоит максимально использовать то время, когда вы просто ходите за кем-то. Задавайте как можно больше вопросов, возможно, не именно в то время, когда ваш наставник создает явно сложный алгоритм, но сразу после того, как он с этим справится.

Благодаря такому вхождению в рабочий процесс можно понять корпоративный стиль компании. Как остальные называют свои переменные экземпляра и методы? Как они дают комментарий? Есть ли единый стиль скобок? Начав с вопросов о стиле, можно вскоре перейти, например, к таким жизненно важным вопросам: как называется сервер базы данных, какие пароли будут мне нужны, какая у проекта структура? Кроме того, вы довольно быстро узнаете своих коллег и поймете, кто хорошо умеет объяснять, а кто не любит, когда в его дела суют нос.

Это не я!

Есть состояние, которое разработчики ненавидят больше, чем любое другое: вы хотите программировать, а внешние обстоятельства вам мешают. Часто эта травма проявляется в школьном возрасте, когда для матери важнее домашнее животное и его потребность пометить участок, чем потребность собственного ребенка спокойно позалипать в мониторе в состоянии потока («Ты уже выгулял собаку?»). Зачастую такую травму просто недолечивают. У вас есть хорошие шансы вызвать именно это состояние. Прямо в ваш первый день! И даже у всех коллег сразу!

«Ты сломал build» — так могли бы звучать слова новоиспеченного коллеги, который хочет ими продемонстрировать, что вы только что лишили всю команду возможности протестировать новый код, то есть практически остановили жизнедеятельность компании и оказались, таким образом, в центре внимания.

А случилось это потому, что вы внесли изменения в центральную систему контроля версий, что вызвало у остальных ошибки компилирования или привело к сбою программы сразу после ее открытия. Эффективная работа после такого вряд ли возможна, потому что большинству разработчиков придется долго проверять, правильно ли работают внесенные вами изменения.

Есть команды, в которых виновного наказывают, надевая на него идиотскую шляпу, которую он должен носить до тех пор, пока кто-нибудь другой не совершит такое же преступление. В других компаниях наказание состоит просто в том, что все, кроме виновного, громко взбивают на кухне молоко для латте до тех пор, пока не появится возможность вновь приступить к работе. Вам нужно предпринять не так уж много, чтобы избежать этой ситуации.

Очевидно, что необходимо как минимум проверить, компилируются ли изменения на вашем собственном компьютере и есть ли желаемый результат. Необходимо также исключить возможность того, что внесенные вами изменения будут иметь нежелательные последствия для абсолютно другой части программы. Как и везде, здесь действует правило: гипотезы — ничто, значение имеют только сухие факты. Даже если вы просто представить себе не можете, что исправление ошибки правописания в обычном вопросе может иметь негативный побочный эффект, все равно тестируйте! Ведь если вы недавно в проекте, то можете и не знать о многих взаимосвязях.

Для того чтобы на всякий случай проверить, вызывает ли изменение какие-либо негативные последствия, многие команды используют автоматизированное юнит-тестирование (см. главу 16). Таким образом можно узнать, к примеру, что казалось бы безобидное исправление орфографической ошибки в тексте выходной функции вызывает ошибку в другой части программы, ошибку, о существовании которой до определенного момента можно даже не знать: парсер ожидает именно этого вопроса с опечаткой. Другой ошибкой, которая точно заблокирует работу всех остальных сотрудников, будет отсутствие проверки необходимых для build данных, поскольку в таком случае файл может быть только на вашем компьютере, но отсутствовать на других, что приведет к неприятному — «а у меня работает» — синдрому. К счастью, в любой системе управления версиями есть функция предоставления списка всех файлов, которые не контролируются программой, то есть могут быть забыты. Соблюдайте правила и в отношении предупреждений, которые выдают система сборки, компилятор или интерпретатор. Даже если эти сообщения предупреждают о ситуа-

ции, которая, как вы знаете, безопасна (скажем, если компилятор замечает, что вы не используете заявленную переменную), вы должны сделать так, чтобы эти предупреждения исчезли перед отправкой кода на проверку в систему контроля версий. Будет неосмотрительно позволять другим разработчикам видеть такие сообщения, потому что этим вы требуете от своих сотрудников, чтобы они привыкли игнорировать предупреждения. Но постоянное игнорирование предупреждений приводит к эффекту привыкания, из-за которого эти полезные сообщения больше не привлекают наше внимание, а значит, мы теряем возможность заранее узнать о багах.

Фактор автобуса

Как и при строительстве дома, при разработке приложения принято распределять задачи между разными членами команды в зависимости от их подготовки, опыта и/или предпочтений. И хотя виды ремесел в сфере разработки программного обеспечения называются не «земляные работы», «кладка стен» и «кровельные работы», а «базы данных», «бэкенд» и «фронтенд», принцип разделения труда тот же. Даже если ответственность не распределяется явно и детально, все равно через некоторое время в команде устанавливается согласие относительно того, какие аспекты программы и какая часть кода принадлежат тому или иному разработчику и кто должен отвечать за содержащиеся там ошибки.

Однако во многих IT-проектах именно эта специализация, то есть неравное распределение знаний об отдельных частях проекта среди участников команды, становится проблемой. Постоянно возникают ситуации, когда отдельные люди оказываются в очень напряженных условиях. Например, если вечером накануне важной презентации для клиентов в программе обнаружился так называемый шоу-стоппер — баг, делающий презентацию невозможной, потому что программа закрывается сразу после старта, выдав на прощание сообщение об ошибке, то разработчику, ответственному за данный участок кода, придется провести бессонную ночь. Другие члены команды могут помочь ему лишь постольку-поскольку, так как у них нет детальной информации о взаимодействии различных частей кода на этом участке. Очень немногие разработчики способны писать в столь напряженной ситуации понятный и правильный код, поэтому, возможно, в эту ночь и получится найти и устранить шоу-стоппер, но качество кодовой базы, скорее всего, пострадает.

> Я сам как-то в начале своей карьеры лихорадочно охотился за багом в коде на C++ для рекламного компакт-диска одного голливудского фильма, а велокурьер, который должен был отвезти окончательную версию в штамповочный цех, уже стоял прямо возле моего рабочего стола и нервно раскачивался на шипах своих велотуфель. В отчаянии я внес в код такие изменения, которые впоследствии казались мне в лучшем случае безобидными, а в худшем — бомбами замедленного действия. Ответственность за программу установки, которая при деинсталляции удаляла не только наше ПО, но и всю информацию с жесткого диска пользователя, нес, к счастью, не я, а другой бледный от ночных бдений эксперт.
>
> *Ян Бёльше, разработчик ПО*

Подобные случаи не исключение, а правило почти во всех IT-фирмах. Однако неравномерное распределение знаний в команде приводит к проблемам не только при поиске ошибок. Болезнь или просто отпуск одного из специалистов также постоянно становятся причинами критических ситуаций и ступора. «К этой модели данных я не притронусь, ее сделал Йорг, и я понятия не имею, что он там напридумывал», — говорит коллега Йорга руководителю проекта, объясняя, почему он не может продолжить работу, пока его коллега не вернется из отпуска и не внесет необходимые изменения. Если проект важен, то Йорг получит в подарок от своего работодателя билет на обратный рейс.

Так называемый *фактор автобуса*, или *фактор грузовика*, показывает, скольких членов команды можно одновременно переехать автобусом без того, чтобы это привело к остановке проекта. Значение варьируется от нуля (самый неблагоприятный случай) до числа всех членов команды. Последнее лучше всего — если не для команды, которую переехали, то во всяком случае для проекта. Под словами «переехало автобусом» подразумеваются, конечно, другие причины выбывания из команды: сотрудники увольняются, заболевают, рожают детей. Для некоммерческих проектов проблемой могут стать даже небольшие изменения в образе жизни участников: конец учебы, влюбленность, расставание, переезд, новые интересы. Если вы не работаете под началом такого руководителя проекта, для которого очень-очень важно совместное владение кодом, то фактор автобуса будет стремиться к нулю. Полностью покончить с этой проблемой нельзя, ее можно только попытаться смягчить. В этом помогут:

- четкое распределение зон ответственности, изложенное в письменном виде;
- четко обозначенные интерфейсы между отдельными частями проекта;
- обзоры кода (см. главу 7);
- баг-трекер;
- совместная работа над кодом.

В фирмах, придерживающихся гибкой методологии программирования, проще распределить знания между большим количеством сотрудников, так как это является частью данной модели процесса разработки ПО. Если вы оказались в фирме, которая не придерживается ни одной модели процесса разработки, то хотя бы позаботьтесь о том, чтобы найти среди своего ближайшего окружения кого-нибудь, кто может встать на ваше место в случае вашего выбывания. Если это возможно с организационной точки зрения, найдите какого-нибудь коллегу себе на замену на тот случай, если вдруг надолго заболеете. Объясните этому коллеге, как организован проект, что необходимо для работы системы, какие задачи по сопровождению требуется выполнять. Записывайте всю информацию о том, что вы сделали и почему. Заносите в систему контроля версий все, что необходимо для компиляции вашей программы, — а лучше всего и ваши записи тоже.

Даже если вы пишете код только для себя и нет никакого заказчика, все равно следует позаботиться о том, чтобы у других была возможность воспользоваться этим кодом, если однажды вы утратите к нему интерес. В идеальном случае этим другим не придется просить у вас разрешения, так как, если речь идет не о невероятно популярном инструментальном средстве, никто не захочет затруднять себя

просьбами. Это значит, что код уже должен вести где-нибудь «публичную жизнь», на Github например (см. главу 21).

Сотрудничество с заказчиками

Большое заблуждение многих разработчиков — такое представление о своей профессии, согласно которому их задачей является разработка ПО, а задачей клиента — предоставление точных сведений о том, что нужно разработать. На самом деле создание программ часто является довольно небольшой частью вашей работы, а львиную долю составляет консалтинг.

Как разработчик, именно вы являетесь тем, кто знает, каким образом можно реализовать идеи в программе, и вы тот, кто должен заметить нереалистичные ожидания и указать на них. Если только вам не выпало счастье работать с технически очень продвинутым клиентом, то придется тратить много времени на то, чтобы выслушивать зачастую расплывчатые и даже порой противоречивые требования вашего клиента и делать реалистичные предложения по их воплощению, так как ваш клиент не может оценить, что возможно, а что — нет.

И даже когда клиенты или пользователи выражаются ясно, они часто имеют в виду не то, что говорят. Их целевая группа непременно хочет, чтобы слева вверху была голубая кнопка, которая будет выдавать какую-то жесть. И вот вы терпеливо создаете голубую кнопку, которая выдает даже две жести одновременно, но при виде нее пользователи говорят: «Нет, мы себе это не так представляли». Не кляните глупость других людей. Так уж устроен мир, и одна из ваших задач в качестве программиста — вместе с заказчиками путем терпеливых уточнений и попыток прийти к результату, отдельные детали которого никто из вас не мог предвидеть.

Сложнее, когда вы работаете не с одним клиентом, а с несколькими представителями компании, чего доброго, еще и из разных, враждующих между собой отделов. Тогда ваши рабочие встречи рискуют превратиться в арену для столкновений; по крайней мере это может привести к тому, что на встречах ваши клиенты будут не столько обсуждать с вами проект, сколько пытаться разобраться между собой и привести свои цели и требования к общему знаменателю. Пока атмосфера дружелюбная и вам платят за присутствие, это не трагично. Вы можете попробовать немного уточнить направление их дискуссии, чтобы к концу все же добиться единства по поводу целей и путей, которыми следует к ним идти. Но если между разными представителями царит напряженная атмосфера, то велик риск того, что проект закончится неудачей, будь вы хоть самым лучшим разработчиком программного обеспечения. Если обстоятельства позволяют, то лучше бросайте это дело и ищите других клиентов. Если нет — думайте о деньгах и не хватайтесь за бутылку: лучшие времена еще наступят.

Работа с добровольцами

Совершенно не обязательно, что вообще найдется кто-нибудь, кто захочет с вами сотрудничать на безвозмездной основе. Если же вам повезло найти таких людей, то стоит ценить это больше собственных профессиональных предпочтений и привычек

и не спорить из-за форматирования кода и других мелочей[1]. Особенно старайтесь не отпугнуть (своей ненадежностью или хамским переписыванием кода и т. д.) заинтересовавшихся некоммерческими проектами. Через несколько лет это будут люди, которые пригласят вас в оплачиваемые проекты или с которыми вы создадите компанию. Ну или нет.

Если в текущем проекте было принято решение, с которым вы не согласны, не стоит уходить, громко хлопая дверью. От искушения поступить именно так может, вероятно, удержать заранее оговоренное правило, что любой, с чьих губ слетит фраза «Если вы хотите делать это так, то без меня!», автоматически вылетает из проекта вместе со своими угрозами. Впрочем, не рекомендуется так поступать, даже если вы чувствуете себя вправе сделать это. Например, вы выступаете за использование прогрессивного формата с открытым исходным кодом, что больше никто не поддерживает или не понимает. Вполне может быть, что мир стал бы лучше, если бы все приняли вашу сторону и установили новые операционные системы, на которых этот формат работал бы. Спокойно предлагайте свой вариант два-три раза. Но если вы и после этого продолжите стоять на своем и тем самым поставите под угрозу дальнейшее существование проекта, то так вы совсем не сделаете мир лучше.

---

[1] Как можно достичь такого спокойного отношения, мы вам, к сожалению, подсказать не можем. В частности, выход этой книги был отложен на год в том числе и из-за долгих споров о том, какой редактор использовать.

ЧАСТЬ III

Работа над ошибками

11 Искусство ошибаться для начинающих

> Его интуиция настойчиво требовала повернуть налево, так как до сегодняшнего дня он всегда поворачивал направо, но та же самая интуиция сказала ему, что его пространственное мышление настолько расстроено, что на саму интуицию нельзя полагаться, поэтому он снова повернул направо.
>
> *Вольфганг Херрндорф. Песок*

Во многих сферах нашей жизни бытует практика рассматривать ошибки как личные недостатки, из которых ничему нельзя научиться. «У меня просто нет хорошего абстрактного мышления», — говорит плохой программист или: «Я просто беспорядочный человек и должен лучше концентрироваться» — и обвиняет самого себя в том, что ему и в третий раз не удалось понять что-то сложное. В то же время программисты склоняются к мысли о том, что пользователь их сайта или автомата по продаже билетов простофиля, а их программу уже невозможно улучшить. В обоих случаях ошибки рассматриваются в качестве единичных случайных недоработок, из которых невозможно сделать какие-то более общие выводы. Однако ошибки не падают с неба, у них всегда есть причина. Эти причины можно выяснить подробнее и таким образом воспрепятствовать появлению этих ошибок в будущем. Ну ладно, не воспрепятствовать, а сделать их менее вероятным.

Программист занимает привилегированную позицию при изучении ошибок: ему гораздо сложнее, чем в других сферах жизни, уклониться от осознания того, что он совершил ошибку. Правда, при написании программ многие ошибки, вплоть до синтаксических, могут долго оставаться ненайденными. Тогда возникает приятная иллюзия того, что программа работает, в то время как на самом деле она лишь ждет, чтобы начать выдавать неправильные значения в периферийных областях либо прекратить работу. И вполне возможно написать синтаксически корректную функционирующую программу, которая в целом станет одной огромной ошибкой. Но все же ошибки на низком уровне в программировании бросаются в глаза быстрее, чем, скажем, во время подиумной дискуссии. Конечно, это не является чем-то уникальным, так как инженеры, пилоты и врачи также быстро замечают, что совершили ошибку в расчетах, сбились с курса или поставили неправильный диагноз. Однако в повседневной жизни мы довольно редко занимаемся разбором своих ошибок. Таким образом, вы сможете накопить опыт правиль-

ного обращения со своими ошибками, который не так-то легко получить в других профессиях.

Еще одним преимуществом для программистов является то, что при возникновении ошибки они точно знают, что речь идет об ошибке, а не о случайной неудаче на бирже, мировом заговоре или произволе каких-либо ведомств. Хотя, конечно, можно еще некоторое время перекладывать вину на другого: компилятор недоработан, сообщения об ошибках появляются из-за каждой мелочи, язык программирования не годится, был интегрирован чужой код, в котором определенно было полно ошибок, или хостер, которому принадлежит сервер, что-то за ночь изменил. Но в большинстве случаев вы рано или поздно все-таки придете к пониманию того, что на самом деле сами допустили ошибку (или десять, или сто).

Скорее всего, самое существенное преимущество ошибок программистов заключается в следующем: в большинстве случаев можно просто нажать на Повторить, чтобы еще раз увидеть их во всей красе. Если бы программные ошибки нельзя было воспроизвести, видимо, им уделяли бы точно так же мало внимания, как и в других сферах нашей жизни.

В гостях у заблуждения: чувствуйте себя как дома

Из истории науки известно, что от научных теорий отказывались не тогда, когда в них накапливались противоречия, а лишь тогда, когда появлялась альтернативная теория, которая могла занять их место. Но и будучи людьми, не имеющими отношения к науке, мы делаем точно так же: придерживаемся какого-либо убеждения до того момента, пока не будет найдено другое. Этот подход применяется в повседневной практике, но из-за этого осознание своей ошибки занимает краткий мимолетный момент.

И здесь программисты в более выигрышной ситуации: осознание своей ошибки в программировании часто не просто занимает очень маленький промежуток времени между прощанием со старым и принятием нового убеждения. Часто часами или днями мы застреваем в ситуации, когда придерживаемся какой-либо ошибочной теории, описывающей реальность, и постоянно позволяем какой-либо неработающей программе водить нас за нос. Осознание ошибки для программиста — не мимолетное состояние, от которого легко избавиться и снова забыть, а скорее состояние, которое будет длиться долго.

Это скорее хорошо, так как лишь при условии того, что человек долго и жестоко страдал из-за своих заблуждений, он будет не так упорно придерживаться своих старых предрассудков и окажется более открытым для новой информации. Можно рассматривать заблуждение как практику медитации: чем больше времени мы проводим с неприукрашенным осознанием того, что только что допустили ошибку, тем лучше для нас — может быть, не конкретно для текущего проекта, но для того, чтобы стать умнее в будущем.

То, какими слепыми нас могут сделать предрассудки, проявляется прежде всего в том, что после решения проблемы часто выясняется: проблема точно описана в документации, в Сети или в другом доступном месте, где вы уже много раз получали

необходимую информацию. Вместе с решением. Жирным шрифтом. Пока вы еще цеплялись за свои ложные убеждения, это решение оставалось невидимым. Хотя ваши глаза увидели проблему и решение, мозг расценил их как раздражающий фактор и поэтому отбросил.

По причине этого именно начинающим программистам следует пытаться развить чувство того, до какого момента можно игнорировать указания на то, что вы допустили ошибку. А потом наступает момент капитуляции, когда вы обессиленно думаете: «Ну ладно...» — и выбрасываете в окно свои старые убеждения. Это момент, когда вы прощаетесь с ложной установкой и начинаете (как минимум на короткое время) точнее и менее предвзято смотреть на вещи.

Часто уже до этого момента появляются гложущие сомнения, которым, однако, вы не хотели поддаваться. Глубоко внутри вы чувствуете, что что-то не так, но несмотря на это еще три раза щелкаете на Перезагрузить в надежде, что проблема исчезнет сама собой. Психологи называют это неприятное ощущение в животе когнитивным диссонансом: вы судорожно цепляетесь за какое-то представление о мире, несмотря на то что факты говорят совсем другое. Пара слабых голосков пищат: «Да это, может быть, совершенно неверно!» — в то время как большая часть хора авторитетно заявляет: «Конечно же, это правильно!» Этот внутренний разлад вызывает неприятные ощущения и может быть прекращен двумя путями: либо вы приспосабливаете свою модель к фактам, либо пытаетесь приспособить факты к модели. То, что вы сначала подавляете сомнение, вместо того чтобы поддаться ему, является не признаком глупости или лени, а попыткой спасти свои предубеждения, изменяя факты в нужную сторону. Этот подход не является неправильным по своей природе, он часто ведет к вполне приемлемым результатам. Факты — и за пределами тоталитарных государств — более гибки, чем о них обычно думают. В этом месте скажем лишь, что вы экономите время, если прислушиваетесь к писку протестующих голосков.

Изучение ошибок в повседневности

Будучи программистом, работающим время от времени, вы не обязательно в четвертый раз распознаете ошибку, которую видели уже три раза до этого. У тех, кто занимается программированием лишь раз в пару недель или месяцев, достаточно времени для того, чтобы все время забывать свои наиболее частые ошибки. Поэтому именно таким программистам полезно сделать поиск ошибок привычкой на всю оставшуюся жизнь. Даже если вы программируете редко, вы можете таким образом упражняться в нахождении ошибок, анализе их причин и устранять или смягчать их последствия.

Вероятно, собственные ошибки при ориентировании в незнакомых городах или сборке мебели от ИКЕА похожи на те, что скрываются за программными проблемами. В конце концов, одна и та же голова ответственна за эти виды деятельности. Тот, кто всегда думает лишь о следующем шаге, вместо того, чтобы планировать заранее, или склонен к решениям в духе «и так сойдет», и в повседневной жизни, как и в программировании, снова и снова будет оказываться с пустым баком на пустынной сельской дороге. Тот же, кто постоянно учитывает все

возможные случайности, будет таскать с собой сумку размером с туристический рюкзак и, предположительно, в программировании попирать ногами принцип YAGNI (см. главу 18).

Если вы однажды правильно подойдете к собственным ошибкам, вам будет легче жить с осознанием того, что другие люди тоже порой не слишком умно себя ведут. Если вы при возникновении ошибок не тратите время на то, чтобы злиться из-за их появления, а думаете над тем, почему они возникли и как избежать их в будущем, вы можете помочь и другим, вместо того чтобы обзывать их и портить им настроение.

Итак, речь идет о том, чтобы при изучении собственных ошибок рассматривать их не как мимолетные незначимые единичные отклонения или просто как вещи, нервирующие вас, а как возможность научиться чему-то. Как и почему появилась ошибка? Можно ли было обнаружить ее раньше? Если да, то как? Существуют ли подобные ошибки в других местах кода или сферах нашей жизни? Возможно ли решить эти проблемы одновременно? Можно ли надежно застраховаться от появления такой же ошибки, а если нет, как можно смягчить последствия этого?

Базу данных съела собака!

Именно когда вы еще не очень хороший программист, то, что вы виноваты в появлении ошибки, гораздо более вероятно, чем то, что это кто-то другой. Поэтому даже с точки зрения статистики имеет смысл сначала взять ответственность за все возникающие ошибки на себя. Таким образом вы избежите ситуаций, когда целыми днями в разговорах с другими вы обвиняли компилятор, язык программирования и других программистов, а в конце концов выяснилось, что ошибку допустили все-таки именно вы.

При этом, возможно, вы будете чуточку быстрее продвигаться вперед, если в разговорах с самим собой и другими будете называть ошибку ошибкой, а не багом. Проблемы — не маленькие пресмыкающиеся, которые проникли в код и ведут там самостоятельную жизнь без нашего участия. Они — созданные нами же самими заблуждения. Как программисты, мы слишком часто включаем в код то, что хотим, не подвергая критической проверке, является ли это тем, что сюда подойдет. У нас есть представление о программном алгоритме, из которого мы хотим отлить программу. Чтобы справиться с этой задачей, мы игнорируем проверку деталей в пользу реализации. А когда программа делает не то, чего мы от нее ожидаем, мы говорим: «Да это баг!» На самом же деле в большинстве случаев мы недостаточно хорошо обдумали, что произойдет, если ожидаемое значение не будет введено, или недостаточно позаботились о том, как должен выполняться цикл, от 0 до `количествоПродуктов` – 1 или от 1 до `количествоПродуктов`. Тот, кто занимается поиском ошибки, считая ее багом, вместо того чтобы искать ее у себя как создателя программы, быстро потеряет терпение, начнет ругаться, упрямо раз за разом запускать программу или обвинять компьютер в глупости. Эти эмоциональные реакции стоят на пути систематического поиска ошибок (см. главу 12). Человек, который рассматривает ошибку в качестве своей собственной, пока не доказано обратное, будет скорее искать в коде, чем обвинять язык программирования или компилятор.

Шлем с мягкой подкладкой

Если вы довольно часто приходите к пониманию того, что сами стали причиной ошибки, то постепенно вы будете становиться немножко умнее. Вы усвоите метазнание, которое выглядит примерно так: «Хотя мои ощущения ведут меня к убеждению, что я все сделал правильно, вероятно, это не так и чувство собственной правоты — лишь часть заблуждения». Через некоторое время вы будете знать, какие ошибки делаете снова и снова, и осознаете прежде всего то, что речь идет о ваших собственных ошибках, а не о жестокости этого мира или неудачах других.

Как не совсем зрелый программист, считайтесь с тем, что вы можете допускать ошибки, так же как люди со светлой кожей пользуются кремом для защиты от ультрафиолетового излучения или скейтбордисты носят защиту на локтях и коленях. Существует столько вещей, которые и опытным программистам, несмотря на поиски решения в Интернете и многочисленные исправления, не удаются на 100 %: форматы ссылок, телефонных номеров, почтовых и электронных адресов или особые символы в иностранных языках изобилуют своими нюансами, поражают своим разнообразием. Собственноручно написанный инструмент категорически не принимает корректные данные, и пользователь выходит из себя. Виной всему убежденность программиста в том, что он предусмотрел все особые случаи. Программист, считающий себя слабым и несовершенным созданием, доставит пользователю меньше неудобств, так как предусмотрит сообщение, которое будет появляться при вводе сомнительных адресов: «Ваш адрес написан в необычной форме. Пожалуйста, проверьте еще раз, не сделали ли вы опечатку. Спасибо!» Во второй раз введенные данные будут приняты, даже если их формат не соответствует представлениям программиста. Таким образом решаются простые проблемы перепутанных полей для заполнения, при этом пользователь с необычными именем и адресом не попадает в руки конкурентов.

В главе 13 приведен список наиболее частых причин ошибок. У каждого дополнительно к этому есть личные заблуждения, которых вы не найдете в этом списке. Полезно создать где-нибудь документ, в который вы будете записывать свои излюбленные ошибки. Так вы, возможно, будете реже их допускать или хотя бы — после просмотра документа — быстрее находить. А если вы еще и столь проницательны, что сразу же станете записывать и возможные решения, то невероятно облегчите себе жизнь.

Отрывок из списка Катрин

- «Важная причина ошибок — игнорировать данный список.
- Иногда недостаточно просто написать что-то вроде `fopen($имя_файла)`, а нужно еще и поместить результат в переменную, иначе ничего не произойдет. А иногда все с точностью до наоборот.
- Что-то работает в PHP4, но не работает в PHP5, или наоборот (а на сервере используются различные версии PHP — в зависимости от того, `cronjob` вызывает скрипт или он вызывается вручную).

- Переменные из-за лени не были инициализированы. Если эта часть программы вызывается много раз, то переменная получает остатки из прошлой операции.
- Цикл `while` и я не дружим. Всегда, всегда, всегда встраивать в него условие прерывания, которое когда-нибудь обязательно сработает. Даже если все выглядит так, как будто это действительно не нужно».

Отрывок из списка Джоанны

- «Перед `svn ci` (внести изменения в систему управления версиями) сначала выполнить команду `svn st`, чтобы посмотреть, что она выдаст. Иначе гарантированно забудешь добавить важные данные с помощью `svn add` и получишь по шапке от коллег за то, что их там нет. Фраза "Так работает же" тоже не делает тебя популярнее.
- Читай проклятую документацию по "Андроиду", Angular или D3, вместо того чтобы все время собирать подходящие кусочки кодов на Stack Overflow.
- В веб-приложениях после обновления файлов CSS или JavaScript, чтобы точно получить самые новые их версии, обязательно обновлять страницу нажатием Shift+R (или Ctrl+R в Internet Explorer).
- Всегда сначала проверять, не оборвалось ли подключение к базе данных.
- В Eclipse при возникновении проблем всегда нажимать Очистить и только потом делать перекомпиляцию. Тогда сообщения об ошибках исчезнут или станут более понятными.
- Работая над веб-проектами, я создаю development- и production-версии, чтобы посетителям сайта был доступен только готовый код. Но, несмотря на это, я уже однажды случайно покопалась в production-версии. С тех пор я всегда снабжаю заголовок страницы пометкой вроде `### DEV ### некое_художественное_творчество ### DEV ###`, которая хорошо заметна как в браузере, так и в текстовом редакторе. Когда я буду переносить код в production-версию, самое плохое, что может случиться, — это то, что я забуду удалить эту пометку. Но это я очень быстро замечу».

Как плохой программист, вы принципиально не должны доверять своей интуиции. Интуиция работает хорошо лишь тогда, когда вы до этого годами подкрепляли ее данными и фактами. Если бы вы были опытным программистом, ваше шестое чувство могло бы быть правильным. Но если вы не опытный программист, а лишь кто-то, кто прочитал книгу Малкольма Гладуэлла, ваша интуиция не права.

В то же время не повредит прислушиваться к своему шестому чувству, пока оно не становится непроверенной базой для построения предположений. Порой у вас без явной причины возникает ощущение неправильности кода. Не будет неверным внимательно проверить этот код с отладчиком или, если такой вариант невозможен, вывести подробные лог-файлы. Это чувство — признак того, что вы постепенно становитесь более хорошим программистом, так как хорошим программистам не нужно кропотливо, команда за командой читать код, чтобы заметить плохие или ошибочные участки.

Для предположений действуют те же правила, что и для интуиции: они так часто вводят в заблуждение неопытных программистов, что в итоге вы экономите время,

не выдвигая никаких предположений. Все возникающие предположения следует тщательно проверять, прежде чем предпринимать что-то, основываясь на них. В особенности это касается предположений о том, что нужно сделать, чтобы оптимизировать код. Как новичку, вам следует взять за правило воздерживаться от оптимизации производительности кода. Если вы действительно считаете, что должны это сделать, воспользуйтесь A/B-тестированием: всегда тестируйте неоптимизированную версию программного модуля, а не ту, которая должна выдать те же результаты, но уже оптимизирована. Если расчеты скорее беглые, следует по обстоятельствам повторить процедуру тысячу раз, чтобы выявить реально измеримые различия.

Избегать предположений I

Мне нужно было написать научное веб-приложение, которое генерировало на стороне сервера картинку, отображающую облако точек. Когда пользователь наводил указатель на одну из точек, браузер должен был показывать всплывающие подсказки с помощью JavaScript. Тексты для подсказок приходили из базы данных в виде списка и вместе с программной составляющей встраивались как скрытые таблицы в код страницы. Это довольно хорошо работало, но только для небольшой части точек — чем больше точек было в облаке, тем реже показывалась подсказка.

После продолжительной отладки выяснилось, что компонент, отображающий подсказки, по умолчанию выводил на экран только 25 значений из каждого текстового списка, а все остальное просто «съедал». Можно было изменить конфигурацию так, чтобы выводились все значения; к несчастью, чтобы это сделать, требовалось установить максимальное значение 0 (да, Oracle, я подозревал тебя). Я предположил: если не указывать максимальное значение, то будет выводиться произвольное большое количество компонентов списка. И действительно, эта несколько неудачная логика со значением по умолчанию 25 была даже упомянута в документации.

Йоханнес

Избегать предположений II

Баг в Ubuntu-Upstart был обнаружен в начале 2010 года: скрипт, который после монтирования жесткого диска должен был очищать временный каталог /tmp, вместо этого без предупреждения форматировал весь жесткий диск, когда любопытный пользователь ради пробы запускал его вручную и без аргументов. Скрипт, или, точнее говоря, автор скрипта, исходил из того, что путь к временному каталогу уже будет корректно считан без проведения проверки, существует ли вообще такой каталог и был ли успешным переход в данный каталог. Если же это не удавалось, скрипт автоматически запускался в корневом каталоге и оттуда форматировал весь жесткий диск.

Образцово спокойный отчет одного владельца жесткого диска о данном баге вы можете прочитать здесь: bugs.launchpad.net/ubuntu/%2Bsource/upstart/%2Bbug/557177.

Избегать предположений в числе прочего означает также учитывать, что ошибки можете совершать не только вы, но и другие. Поскольку окружающие не всегда

делают то, чего мы от них ожидаем, данные, которые мы запрашиваем из Интернета или из программы, приходят к нам не так быстро, как бы того хотелось. Иногда адресат вообще не отвечает. Функции и библиотеки работают не так, как написано в комментариях или документации к ним. Пользователи ведут себя не так, как поступали бы мы.

Поэтому рекомендуется делать минимум предположений и исходить из того, что все, что может пойти не так, когда-нибудь пойдет не так (точно так же, как и вещи, которые, собственно, совершенно не могут пойти не так). В связи с этим прочтите главу 16 и напишите тесты для ваших предположений, которые помогут обнаруживать ошибки в них на раннем этапе.

12 Отладка I: поиск ошибок как наука

> «Разработчик ПО». Это словосочетание вызывает в вашем мозгу следующую картину: у вас есть запутанный клубок кода, который вы затем распутываете. В большинстве случаев это даже верно.
>
> *Ян Варвиг/@agento, Twitter, 4 августа 2013 года*

Тот, кто путешествует, рано или поздно заблудится. В литературе много веселых и жутких историй, в которых люди сбиваются с пути и своими необдуманными действиями еще сильнее загоняют себя в угол. В связи с этим клубы путешественников и дирекции национальных парков дают путешественникам пару хороших советов, каким образом в такой ситуации можно снова выбраться в безопасное место или хотя бы не ухудшить свое положение.

Разработка ПО, как правило, гораздо менее опасна, чем поход в горы. В процессе разработки нельзя умереть от жажды, редко нужно спать в промокшей палатке, да и во время внезапно наступившей непогоды гораздо комфортнее провести время за письменным столом, чем на высокогорном пастбище. Однако есть и то, что объединяет оба этих типа деятельности, — движение по непросматривающейся, а часто и не очень знакомой местности. Чтобы достичь цели, перед путешествием вам нужно мысленно представить окружающую вас среду, при разработке ПО вы должны смоделировать в голове программу и ее поведение. Ошибки нередко ведут к большой путанице, так как ваши представления о мире и реальные условия больше не совпадают.

И ошибки при разработке являются не исключением, а правилом. В ходе различных более ранних исследований[1] пришли к выводу, что программисты делают ошибки в 2–5 % строк кода, и эти исследования опираются на данные, полученные при тестировании опытных разработчиков.

Не все советы для путешественников применимы в работе с программными ошибками. Так, разработчику ничем не поможет совет сообщить хозяину гостиницы конкретное время и цель похода, так как, если вы запутались в коде, никто не вышлет спасателей. Однако во время подготовки к путешествию и при разрешении

---

[1] См. общий обзор здесь: panko.shidler.hawaii.edu/HumanErr/ProgNorm.htm.

сбивающих с толку ситуаций, случающихся во время программирования, есть несколько основных правил, которые могут облегчить вам жизнь.

Планируйте на перспективу. В особенности позаботьтесь о том, чтобы вы всегда могли снова вернуться в исходную точку

Прежде чем вы тронетесь в путь и начнете переписывать какой-либо участок кода или устранять упрямую ошибку, вы должны подготовиться. Используйте систему управления версиями (см. главу 21) и сохраните код, прежде чем вносить большие изменения или начинать исправление сложных ошибок. Только таким образом вы сможете в любое время вернуться в исходную точку. Если выработаете в команде, вам, естественно, не следует сохранять заведомо неправильный код в общем хранилище. В этом случае вам следует использовать возможности системы управления версиями и создать ветви и ваш личный «отросток», в котором вы будете исправлять определенные ошибки. Как только ошибка будет устранена, ветвь снова будет интегрироваться в основное направление разработки.

Мы рекомендуем срочно начать применять систему управления версиями, но если вы не можете себя преодолеть и начать пользоваться ею, сделайте одолжение и создайте резервную копию данных, над которыми вы хотите работать. Если вы уже знаете, что вам не удастся каждый раз принуждать себя и к этому, тогда побеспокойтесь об автоматическом создании резервных копий на сервере и/или локальном компьютере. Таким образом, вы все-таки еще сможете вернуться к вчерашнему состоянию. Как это происходит, описано в главе 21.

Заметьте место, на котором вы потеряли ориентацию. Не настаивайте на том, что все в порядке или по крайней мере скоро снова будет в порядке

Большинство программных ошибок легко заметить и исправить, часто речь идет о простых ошибках в орфографии или синтаксисе. Напротив, логические ошибки, не приводящие к вынужденной остановке работы программы, а изменяющие значения данных, можно найти и исправить лишь с огромным трудом. Именно как неопытные программисты, мы часто теряем ориентацию и наобум исправляем непредполагаемые ошибки, часто сопровождаемые мыслью: «Я же все сделал правильно». В этом месте люди часто становятся суеверными: мы настаиваем на том, что на самом деле все понимаем, только компилятор, база данных и операционная система якобы замыслили заговор против нас. На этом месте отступите назад и признайте, что вы в действительности понятия не имеете, что сейчас перед вами происходит. Это освобождает путь для систематического поиска ошибок, который мы описываем далее.

Если потеряли ориентацию, не впадайте в панику, так вы только ухудшите свое положение

Если в процессе программирования у вас совершенно ничего не получается, не пытайтесь силой прорваться к цели. Как только в процессе поиска ошибок вас охватывает фрустрация, злость или скука либо вы поддаетесь суеверным идеям, вы очень легко наносите ущерб больший, чем изначальная проблема. Это состояние, в котором вы склонны случайно удалить базу данных или сбросить ноутбук со

стола. В нашем примере риску подвергается лишь код, но то же самое помутнение рассудка ведет к тому, что заблудившиеся путешественники из-за ярости, которая не позволила им посмотреть на карту, срываются в расщелину. В обоих случаях безобидная проблема превращается в серьезную.

Не все так плохо в состоянии беспомощности: все ваши взгляды и суждения о причинах возникшей ошибки, существовавшие ранее, уже основательно разбиты на кусочки. В данный момент вы не держитесь за такое большое количество предубеждений, как обычно, и можете — теоретически, если вы не слишком заняты своей злостью, — более искренне воспринимать то, что действительно происходит в вашем коде или окружающей вас среде. Любителям жанра ужасов знакома ситуация, когда запачканная кровью главная героиня после всех мучений останавливается и приходит к выводу, что ее предшествующая стратегия была не особенно успешной. В данный момент она становится другим человеком, тем, кто находит выход из ситуации там, где прежде его не было. Несмотря на то что фильмы ужасов не слишком полезны в повседневности, свободной от зомби, в данном случае они предлагают настоящую помощь: негибкие мнения и неуместные стратегии мешают нам распознать в новых ситуациях то, что может помочь.

Начать разбираться в тяжелой ситуации и не встретить ее ни чистым насилием («Я сейчас просто буду долбить этот код, пока он не заработает, хоть всю ночь!»), ни маневрами уклонения («Что происходит с этим "Твиттером"?») тяжело. Минимум людей охотно признают, что они, возможно, совсем не так компетентны, как это выглядит на фотографии в их резюме. Менее опытные программисты здесь снова в выгодном положении: вы не должны обманывать ни себя, ни мир, говоря, что у вас все под контролем. Вы можете расслабиться и с мыслью: «Это программирование, совсем не так просто!» — принять свою растерянность.

Систематический поиск ошибок

> Многие не понимают, что уход за мотоциклом — абсолютно рациональный процесс. Они думают, это что-то вроде сноровки или что у кого-то «руки приспособлены к машинам». Это верно, но сноровка почти полностью состоит из логического подхода, и большинство проблем возникают из-за «короткого замыкания между наушниками», как говорят старые радиолюбители, — неспособности правильно использовать свою голову. Мотоцикл полностью подчиняется законам логики. Поэтому нужно, так сказать, сдать экзамен на здравый смысл, чтобы научиться ухаживать за мотоциклом.
>
> *Роберт М. Пирсиг. Дзен и искусство ухода за мотоциклом*

Ненаучный поиск ошибок в основном состоит из попыток наобум что-нибудь изменить и попробовать, не заработает ли сейчас. В голове программиста при этом крутится нечто вроде: «Результат какой-то слишком маленький, сложная штука это исчисление процентов. Может, стоит все-таки в конце умножить на сто? Да, так лучше. Ура, рабочий день закончился!»

Переписывать код на основе простых предположений редко оказывается хорошей идеей. Во-первых, при этом вы, вероятно, допускаете новые ошибки. Плохие

программисты могут сразу же уяснить, что на одну исправленную ошибку допускают одну-две новых. Если изменение кода происходит на базе обычного предположения, высока вероятность того, что это предположение ложно и вы лишь встраиваете в код новые ошибки, не решая исходную проблему. В таком случае вы делаете не два шага вперед и один назад, а лишь шаг назад.

Во-вторых, надежно исправить ошибку можно лишь тогда, когда найдена ее причина. Тот, кто терпеливо ковыряется в коде и пробует изменения, вероятно, когда-нибудь устранит ошибку. На данный момент он даже ее действительно устранил. Более вероятно, однако, что он просто устранил симптомы или переместил ошибку в другое место. Через пару недель или месяцев старая ошибка всплывает снова в новом виде, и поиск начинается сначала.

Поиск ошибок — хороший способ тренировать научный подход. Первый шаг — точные наблюдения. Что там вообще происходит? Что является ошибкой и что, собственно, должно было бы произойти? При каких условиях появляется ошибка? На этой стадии вы восприняли и определили проблему. Второй шаг предполагает построение гипотезы того, как данная ошибка появляется. А на третьем шаге тестируете эту гипотезу, исправляя предполагаемый неправильный участок кода и проверяя его в условиях, при которых появлялась ошибка. После этого либо ошибка устранена, либо вы снова начинаете со второго шага и выстраиваете новую гипотезу. При этом именно забывчивым и неопытным полезно вести протокол отладки, записывая все, что они сейчас делают. Если же вы пытаетесь запомнить все, не ведя записи, уже через полчаса у вас появится навязчивое подозрение, что вы, как заблудившийся, ходите по кругу и пробуете в точности то же самое, что пробовали в самом начале.

> Всегда неприятно, когда ничего не записываешь, а потом, например, уезжаешь на неделю в отпуск. Возвращаешься и думаешь: «Что же тут, собственно говоря, нужно делать? Ага, не работает, где же ошибка?» Все это только для того, чтобы целый день искать ошибку и потом прийти к выводу, что это было именно то, что ты делал весь день перед отпуском, а именно: целый день искал ошибку, нашел ее, но не записал, что же было ошибкой.
>
> Во всем, что не касается компьютера, я веду себя так же, у меня с собой всегда есть блокнот. Обо всем, что как-то подходит под определение рекурсии, или трансформации координат в координатной системе, или подобном, я действительно от руки делаю записи. Либо до, тогда в большинстве случаев все работает, либо, если не работает, после.
>
> *Корнелия Травничек, разработчик ПО и автор*

Наблюдение

Наблюдение, возможно, является одним из самых недооцененных научных методов, ведь просто понаблюдать — это что, настолько сложно? На самом же деле на наблюдения накладывают четкий отпечаток наши ожидания, а не учитывать ожидания может оказаться сложной задачей, так как при наблюдении мы бессознательно формируем гипотезы увиденного. В большинстве случаев это полезно, так как из-за простой опечатки вам не нужно сидеть полчаса и смотреть на код как баран

на новые ворота, но при поиске сложных ошибок эта связь наблюдения и поиска причин порой может стать препятствием.

Отчет об ошибках в браузере Opera является в данном плане показательным (рис. 12.1).

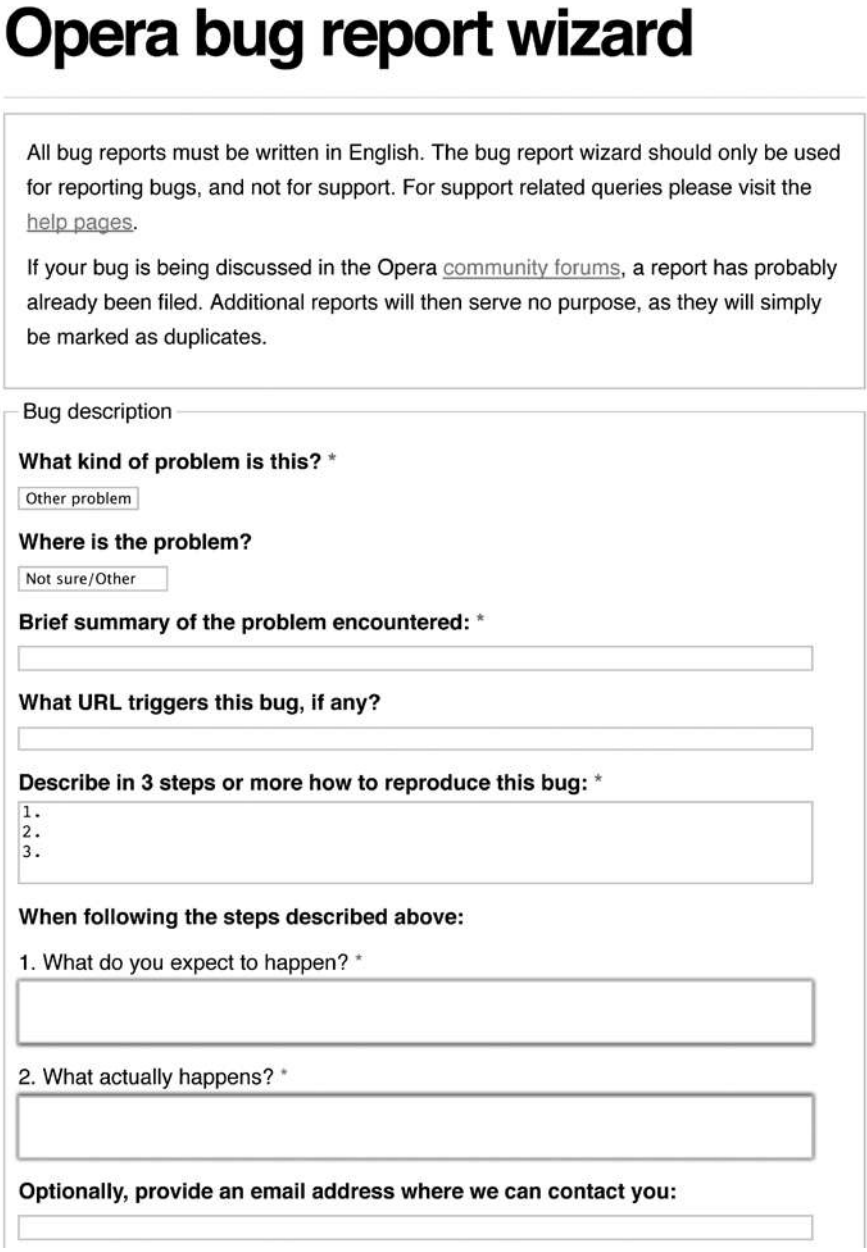


Рис. 12.1. Пример формы для сообщения об ошибках, Opera

- О проблеме какого рода идет речь? (Вынужденная остановка, пробел в безопасности...)
- Краткое описание (одна строка).
- Какие внешние условия (в данном случае какие адреса веб-страниц) вызывают проблему?
- Краткое пошаговое описание того, как можно воспроизвести проблему (три шага).
- Какой результат вы ожидали?
- Что произошло на самом деле?

У вас нет необходимости всегда подходить к этому так бюрократически, но вначале будет полезно распечатать данный список и положить возле клавиатуры. Если отыщете ошибку — заполните его. Возможно, уже на данном шаге или когда будете просматривать записи позже, вам бросится в глаза, что ошибка еще недостаточно четко описана. Потратьте время на то, чтобы еще раз просмотреть описание.

Кроме того, будет полезно завести с краю протокола отладки графу для непривычных явлений, в которую будете записывать то, что совершенно не зависит от преследуемой вами ошибки, но происходит не так, как всегда, от «сервер реагирует медленнее, чем обычно» до «всегда, когда я запускаю программу, в здании включается пожарная сигнализация». В конце снова и снова будет выясняться, что эти наблюдения были совсем не так уж не связаны с текущей проблемой, как вы сначала думали.

> JPEG-парсер, который работал для камеры охраны, всегда прекращал работу, когда к двери фирмы подходил генеральный директор. Ошибка повторялась в 100 % случаев. Без шуток. Для всех, кто не так уж много знает о сжатии данных в JPEG: картинка разбивается на что-то вроде матрицы, составленной из маленьких блоков, которые кодируются с помощью магии, и т. д. Парсер каждый раз давился, когда приходил директор, потому что тот всегда приходил в клетчатой рубашке, которая приводила к возникновению какого-то особенного случая из контраста и границ блоков.
>
> *stackoverflow.com/questions/169713/whats-the-toughest-bug-you-ever-found-and-fixed*

Чем больше данных вы соберете на данном шаге, тем лучше. Используйте также различные формы представления данных, какие только возможно. Гипотеза, которая никогда не пришла бы вам в голову, если бы вы смотрели на данные в виде текста, может сразу же броситься в глаза при их графическом представлении.

Что усложняет наблюдение

Наблюдения стремятся стать гипотезами

Это отнюдь не так легко — протоколировать только то, что действительно видишь. Нетренированный наблюдатель видит мужчину, бегущего за автобусом, а в действительности этот мужчина просто бежит в том же направлении, что и случайно проезжающий мимо автобус. Абсолютно свободное от гипотез наблюдение невозможно,

не будьте слишком строги к себе. Каждое наблюдение связано с какими-то предположениями, в нашем примере вместо мужчины речь могла идти о женщине, инопланетянине или симуляции матрицы. Однако запись и позднейшая ее оценка могут привести к новой, не предполагавшейся ранее гипотезе или же перечеркнуть ее.

Избирательное восприятие

В конце поиска ошибки часто выясняется, что, собственно говоря, с самого начала вокруг вас находилось множество огромных указательных знаков с надписями «К ошибке, пожалуйста, вот сюда!». И так не только в программировании. В этом виноват в основном феномен слепоты невнимания, который лучше всего объяснить на примере классического эксперимента с гориллой, проведенного в 1990-е годы. Испытуемым показывали короткое видео с записью баскетбольной игры и просили их посчитать количество передач. Примерно половина зрителей из-за концентрации на мяче не замечала, что в середине видео через площадку проходил человек в костюме гориллы. В процессе отладки количеству передач соответствует поиск определенных симптомов, которые, как мы предполагаем, указывают путь к ошибке. Другие проблемы, такие же большие и волосатые, как горилла, могут при этом проскользнуть через экран незамеченными.

Когнитивный диссонанс

Желание устранить ошибки в написанном вами коде является лишь одним из многих, которые наш мозг должен взвешивать, соотнося друг с другом. В числе прочего оно конкурирует с желаниями считать себя правым, никогда не совершать ошибок, производить на других людей впечатление умного человека и не быть вынужденным слишком сильно обдумывать что-либо. Если одно из этих желаний берет верх, то наш мозг прикладывает значительные усилия, чтобы убрать все указания на ошибку или ее причины из нашего восприятия. Если положение дел позволяет нам казаться хорошими программистами, мы благодарно принимаем его услугу. Если же факты приводят нас к заключению, что мы что-то сделали неправильно, мы смотрим в другую сторону.

В связи с этим практический совет от Чарльза Дарвина: «В течение многих лет я следовал золотому правилу, а именно: когда я сталкивался с опубликованными фактом, наблюдением или идеей, которые противоречили моим основным результатам, я незамедлительно записывал это: опытным путем я выяснил, что такие факты и идеи гораздо легче ускользают из памяти, чем благоприятные». И в этом вам поможет протокол отладки, а еще лучше — общий протокол к коду, в который вы заносите все необычные вещи, перед тем как отладка вообще оказывается необходимой. Бороться с тем, что ошибка остается незамеченной, так как вы смотрите на код с большего расстояния, помогает также метод утенка (см. главу 13).

Анализ и построение гипотез

Если вы выделили и по возможности записали проблему, следующим шагом станет построение гипотез, которые могли бы объяснить ошибку. Это может быть совершенно банальным: «Хм-м, даже если ошибка *в* этой функции, то входное значение долж-

но быть таким и только выходное значение неправильно. Но если ошибка *до* нее, то и входное значение уже другое. Итак, проверить еще раз входные и выходные значения». У вас уже есть две конкурирующие гипотезы и тест, чтобы выбрать одну из них.

Стройте и проверяйте гипотезы одну за другой, даже если вам хочется сделать две вещи сразу: так будет быстрее и у вас все равно уже нужная строка перед глазами. Это довольно сложно — ввести в код одно изменение и не создать новых проблем. Если же вы изменяете несколько мест за один раз, то очень легко создаете совершенно новую проблему, последствия которой похожи на изначальную ошибку. При этом новая и старая ошибки даже не должны быть особенно похожи — неопытные программисты так быстро забывают, какую проблему они, собственно, хотели устранить, что охотно думают, что совершенно другая, новая ошибка и была старой. Это конец научного метода и начало совершенно новой запутанной ситуации: вы просмотрели ошибку, а может, вы ее уже устранили, но вместо нее вписали другую с новыми последствиями или же вы все-таки не нашли старую ошибку и в дополнение внесли новую? Гораздо более затратно по силам, но более четко и в итоге экономно по времени проводить отладку маленькими шажками и после каждого шага проверять результат.

Если вам в голову приходит гипотеза, в то время как вы проверяете другую, запишите новую идею и отложите блокнот в сторону. Проверьте до конца текущую идею, прежде чем посвятить себя блестящей новой. Если ошибка появляется в определенных массивах данных, запишите эти массивы и парочку других, в которых ошибка не появляется, в качестве тестовых.

Что усложняет построение гипотез

Предубеждения

Мы склонны к тому, чтобы обвинять в наших ошибках других (людей, предметы или программы). Если вы как программист уже однажды нашли в определенном модуле ошибку, в будущем очень скоро начнете винить этот модуль, если что-то пойдет не так. Это имеет определенный смысл, поскольку ошибки, как говорит опыт, не появляются поодиночке, но это подтолкнет вас к поспешным заключениям. Мы склонны видеть ошибку в этом модуле и не выстраиваем никаких альтернативных гипотез, которые, например, могли бы сказать нам, что в подозрительном модуле ошибка лишь проявляется, но на самом деле ее причина в другом.

Гипотезы стремятся стать убеждениями

Описанный в главе 11 разрыв между двумя убеждениями — плодотворное состояние. Но в нем крайне неуютно и велико искушение как можно скорее снова вернуться на кажущуюся твердой почву нового убеждения. Соответствует ли данное утверждение наблюдаемым фактам, часто является второстепенным. Гипотезы неудобны для нашего сознания, и они тайно становятся убеждениями, если мы активно не препятствуем им в этом. Осознать данную тенденцию полезно не только для отладки ПО, но и для того, чтобы правильно ориентироваться в политических темах, на местности и в человеческих отношениях.

Проверка гипотез

Наш заблудившийся путешественник из начала главы тоже строит гипотезы о своем местоположении и проверяет их в реальности, когда ищет на карте какие-то ориентиры или, например, идет вдоль гребня горы, чтобы сравнить очертания следующей долины с картой. Если он особенно терпелив и/или опытен, он также смотрит в другом направлении, а именно следит, соответствует ли данный ландшафт обозначенному на карте. Это ослабляет искушение посчитать смутно похожую местность искомой. При отладке все точно так же: проверка гипотез происходит в процессе сравнения различных версий программы с желаемым результатом.

Сначала проверьте ваши грубые предположения о том, где появляется ошибка. Только когда установили *«где»*, вы должны переходить к *«почему»*. Если какая-то функция получает еще правильные значения, но выдает ложные результаты, то эта или другая функция, на которую ссылается данная, ошибочны. В таком случае попытайтесь ограничить область функции, в которой появляется ошибка, в зависимости от предпочтений с помощью команды `print` или журналов (см. главу 13). Выполните разбивку на блоки: прописывайте команду в виде цикла, оператора `if/else` или вызова функции перед каждым блоком кода и еще одну команду — после него. В наиболее сжатом виде это выглядит так:

```
function do_something() {
    print "i was here 1";
    if (foo == bar) {
        print "i was here 2";
    } else {
        print "i was here 3";
    }
    print "i was here 4";
    return something;
    print "look, i was even here 5";
}
```

Если вы выделили блок с ошибкой, то можете начать формулировать и проверять гипотезы на предмет причин ошибки: «Ага, может быть, цикл проводится слишком часто, зададим-ка мы для пробы счетчик».

Вполне возможно, что вам придется написать тестовый код или создать тестовые данные, чтобы проверить разработанные гипотезы. При простых ошибках, которые ведут к вынужденному прекращению работы программы, тестовый код и данные не очень важны, поскольку картина ошибки и так понятна. Напротив, если вы пишете программу для оценки больших объемов данных — например, в научных целях, — у вас часто возникают проблемы недостаточной прозрачности задачи и/или нехватки времени. Если вам, например, нужно обрабатывать иерархические структуры данных с несколькими слоями или если объем данных настолько большой, что проверка длится несколько минут, вы не зря потратите время, если специально создадите маленький упрощенный массив данных.

Что усложняет проверку гипотез

Склонность к подтверждению своей точки зрения

Как выяснил в опубликованном в 1960 году эксперименте английский психолог Питер Каткарт Васон, люди склонны к тому, чтобы подтверждать свои предположения, и избегают сомнений по их поводу. Это явление было названо по-английски confirmation bias, или поиск подтверждения. Участники его эксперимента получали последовательность из трех чисел: 2, 4, 6. Их задачей было понять, какая закономерность лежит в основе данного числового ряда. Кроме того, они показывали экспериментатору собственные три числа и получали замечание о том, соответствуют они закономерности или нет. Если бы этого не наблюдалось, участники должны были бы поступить следующим образом.

1. Посмотреть на последовательность чисел и построить гипотезу о том, какая закономерность скрывается за ней, например: «все время прибавлять два».
2. Написать три числа, удовлетворяющие данной последовательности, и показать их экспериментатору (пример верификации): «12, 14, 16».
3. Написать три числа, которые противоречат установленной закономерности, и показать их экспериментатору (пример фальсификации): «10, 11, 12».
4. Если пример верификации был отклонен экспериментатором или был принят пример фальсификации, отбросить старую гипотезу и построить новую закономерность.

Однако практически все участники пропустили шаг 3. Прежде всего они показывали экспериментатору такие последовательности из трех чисел, которые опирались на выбранную ими закономерность, и лишь очень редко — годящиеся для фальсификации их предположений. Так как закономерность в задании выглядела как «увеличивающиеся числа», а это означало, что предложения типа «1, 37, 948 563» тоже были бы приняты, участникам эксперимента было очень сложно дойти до правильного решения. Поиск подтверждения наших предположений может отпугнуть нас от установления жестких критериев, на основании которых мы были бы готовы отказаться от своих убеждений.

Пренебрежение отрицательными результатами

В науке снова и снова обсуждается проблема того, что научные журналы слишком охотно публикуют сообщения об исследованиях с положительными или значительными результатами и слишком редко — об исследованиях, которые не принесли никаких результатов. Это так называемая склонность к подтверждению написанного в публикации. Однако исследование, из которого следует, что средство X не оказывает никакого влияния на болезнь Y, также важно для науки, и если оно исчезнет неопубликованным в ящике стола, истинная картина исказится. В отладке все очень похоже. Если для проверки подозрения вы проводите тест и этот тест не находит ошибку, есть две вероятности: или тест не делает того, что должен, или программист не совсем понял, что является проблемой, и проводит тест не в том месте. Незначительный результат может быть таким же полезным при поиске пути к устранению ошибки, как и на первый взгляд гораздо более

правильный и однозначный. Будьте хорошим ученым, внимательно относитесь и к отрицательным результатам.

Пренебрежение неподходящими ответами

Результаты, которые не вписываются покорно в наше представление о проблеме, мешают нам — время от времени они нас даже раздражают. Мы склонны игнорировать их, но это ошибка. Каждый факт, который не соотносится с ожиданиями, передает нам важное сообщение, а именно то, что либо мы не совсем правильно поняли проблему, либо у нас сложилось совершенно неправильное представление о ней.

> Дайте проблеме время. Смотрите на нее как на леску на рыбалке. Пройдет совсем немного времени, и вы почувствуете легкое подергивание, небольшой скромный результат, который вас робко спрашивает, интересно ли вам что-либо в таком роде. [...] Покажите свой интерес. Попытайтесь сначала понять новый результат не в отношении вашей большой проблемы, а ради него самого. Может быть, проблема не так уж и велика, как вы думаете. И может быть, результат не такой уж и маленький, как вы думаете. Может быть, это совсем не тот результат, который вам нужен, но отбросьте его только тогда, когда полностью уверены в этом. Если вы не будете отбрасывать его сразу, то часто станете замечать, что поблизости у него есть друзья, которые ожидают вашей реакции. И среди этих друзей, возможно, именно та информация, которую вы ищете.
>
> *Роберт Пирсиг. Дзен и искусство ухода за мотоциклом*

«Научный подход» — это звучит намного проще и намного более упорядоченно, чем есть на самом деле. Даже если у нас есть совершенно конкретная заинтересованность в том, чтобы найти ошибку в своих мыслительных операциях и в нашем коде, часто мы сами стоим у себя на пути. Это не личный изъян — и совершенно не один из тех, которые присущи только неопытным программистам, — а общая проблема всех человеческих голов. Ни ошибки, ни выбор окольных путей при их устранении не возникают просто так или по глупости. Полезно понять, что при этом происходило в вашей голове.

13 Отладка II: найди ошибку

А сейчас распечатай исходный код, запеки его с сыром — я надеюсь, что так все сбои, которые остались, будут устранены.

Тобиас Фибигер/@scholt, Twitter, 7 августа 2010 года

Сообщения об ошибках — наши друзья

Отладка должна иметь как можно меньше общего с предположениями, гаданием и отчаянием от изменения случайных частей программы. Поэтому используйте базу данных, браузер, веб-сервер и иные отдельные части программы, чтобы вывести как можно более точные состояние переменных и хронологический ход выполнения программы, так как с помощью результатов отладки часто можно проследить ход выполнения до появления какой-либо ошибки. Это начинается со сведений, которые выдает вам компилятор, — сообщения об ошибках и предупреждения.

Даже если предупреждения сначала производят впечатление надоедливого шума и кажутся не такими важными, как сообщения об ошибках, следует серьезно воспринимать их и устранять вызывающие их проблемы, так как, во-первых, после многочасового поиска ошибки может выясниться, что именно предупреждение, которое вам все время казалось неважной мелочью, было ключом к ошибке. Во-вторых, в противном случае вы привыкнете к предупреждениям и начнете их игнорировать. К чему это ведет, вы можете заметить на примере различных нерадостных событий на атомных электростанциях, при запуске ракет. Предупреждения не простые придирки, даже если неопытные программисты часто их так интерпретируют. Разработчики языка включили их в свой язык, чтобы обратить внимание на не совсем чистые приемы и возможные ловушки, и нужно признать, весьма вероятно, что разработчики языка знают о языке больше, чем мы. Поэтому предупреждения — это предложения, которые вы не должны легкомысленно отбрасывать.

Даже если вы принципиально ничего нигде не читаете, стоит узнать, как вы можете в выбранном вами языке повернуть краники на счетчиках ошибок направо. Если это приводит к тому, что программа, которая прежде работала безупречно, сейчас выдает огромное количество сообщений об ошибках, это не повод для отчаяния. Количество ошибок совершенно ничего не говорит о том,

насколько серьезно положение в действительности. Одна-единственная ошибка может послужить причиной серьезных сбоев во всей системе, и это несмотря на то, что вы храбро пытаетесь что-то сделать с кодом. Начальный сбой порождает новые сообщения об ошибках на других страницах. Исправьте первоначальную ошибку, тогда исчезнут и ее последствия в виде новых ошибок. Если вы столкнулись с лавиной сообщений об ошибках, сконцентрируйте внимание на первой из них.

К сожалению, далеко не во всех языках сообщения об ошибке действительно описывают ошибку. Особенно часто строка кода, в которой якобы появляется ошибка, указана неправильно, так как компилятор жалуется на ошибку лишь в том месте, в котором ее заметил. Например, если вы забыли открывающую скобку, компилятор ничего не заметит, однако он увидит одну лишнюю закрывающую скобку и будет обвинять ее. Указанный в сообщении об ошибке номер строки — самое последнее, что может подойти: ошибка может быть в ней, может быть задолго до нее, но никогда — в последующих строках. Часто ошибка именно в предыдущей строке, так как там, например, вы забыли поставить точку с запятой. Ошибки, якобы находящиеся в самой последней строке, указывают на отсутствующую фигурную скобку. Эта скобка может быть где угодно в коде, находящемся выше, поэтому мудрым решением будет использовать редактор с функцией *выделения скобок*[1].

Кому тут что-то от меня нужно?

Сообщения об ошибках могут появляться из разных источников. Синтаксические ошибки отслеживаются парсером — программой, распознающей отклонения от правил синтаксиса данного языка. Если вы набираете `prnit`, то ваш редактор (если у него есть парсер для используемого вами языка) или парсер в *компиляторе/интерпретаторе* установит, что не знает такого слова. После чего он проверит в связанных с ним библиотеках и коде, не описано ли где-нибудь значение данной команды. Так как речь идет об опечатке, он ничего не найдет и сообщит вам о синтаксической ошибке. Поэтому синтаксические ошибки являются очень простыми: вы сразу же их найдете, причем гарантированно. Хорошие редакторы и среды разработки особенно полезны для исправления таких ошибок, так как они находят их уже в процессе написания. В языках программирования, созданных для определенного узкого типа задач (см. раздел «Типы переменных» главы 26), указание неверных типов переменных (строчная переменная там, где ожидалось число) также ведет к ошибкам, о которых программа вам сообщит еще в процессе программирования.

---

[1] Если непонятно, в каком документе отсутствует скобка, вам поможет способ решения, который работает и в самых простых редакторах: откройте все документы и ищите в них с помощью функции, которая в большинстве случаев выглядит примерно как «найти во всех открытых документах», { и }. Если скобки действительно нет, у вас будет разница в 1. Теперь наобум закройте пару документов, еще раз пересчитайте. Если разница исчезла, ошибка в одном из закрытых документов. Затем вы повторяете данную процедуру с этими документами до тех пор, пока не найдете место, где отсутствует скобка.

КОМПИЛЯТОР И ИНТЕРПРЕТАТОР

Компилятор переводит исходный код в рабочую программу, которую можно запустить на другом компьютере, на котором не установлен компилятор. Интерпретатор считывает ваш код и выполняет его, однако он должен быть на таком же компьютере.

Ошибки компиляции появляются в тех языках, в которых существует четкая граница между процессом компиляции и запуском программы. Хотя сегодня большинство языков компилируется (раньше было гораздо больше языков, которые лишь интерпретировались), наиболее распространенным методом является компиляция «точно в срок» — исходный код компилируется лишь тогда, когда он должен обрабатываться. В более старых языках и языках, ориентированных на высокую производительность, процесс компиляции, как правило, происходит эксплицитно. Компилятор обрабатывает исходный код, определяет, какие еще ресурсы необходимы (библиотеки, другие файлы-источники), проверяет, существуют ли они, а затем переводит все в *машинный код*. Если вы сослались на библиотеку, но, например, указали неправильный путь, компилятор остановит перевод программы и выдаст ошибку.

МАШИННЫЙ КОД

Машинный код действительно поступает в процессор и обрабатывается там, поэтому программа может работать только на процессоре, для которого она компилируется. У каждого семейства процессоров свой машинный код, у процессоров Intel не такой, как у процессоров ARM в смартфонах или процессоров старых компьютеров фирмы Apple.

> Сейчас я очень приветствую те языки, где есть компилятор, который может проанализировать программу и сказать мне еще перед тем, как что-то будет запущено, где ошибка. Как-то я перешел на такой, с Ruby на Scala, и я не хочу назад, так как у меня теперь есть собеседник в виде компилятора, который мне очень сильно помогает: он говорит мне, где я сделал очевидные ошибки. Раньше у меня такого никогда не было, и именно мне приходилось постоянно кропотливо проверять, где же что-то пошло не так.
>
> *Лукас Харманн, разработчик ПО*

Ошибки времени выполнения, напротив, бросаются в глаза лишь в ходе выполнения программы и обычно приводят к внезапной вынужденной остановке программы, если разработчик не препятствует этому подходящей обработкой ошибок (см. раздел «Обработка ошибок» главы 26). В данном типе ошибки ресурсов возникают из-за неправильной конфигурации (вы указали неправильный сервер баз данных в конфигурации) и непредусмотренных событий (целевой сервер в данный момент не работает). Однако большинство ошибок времени выполнения являются ошибками программиста, как, например, следующие.

- Вы не инициализировали переменную, но хотите ее применить. В языках с эксплицитным процессом компиляции (например, C++ или Java) эта ошибка в большинстве случаев уже будет найдена к моменту компиляции, в динамических языках, например в JavaScript или Ruby, — только к моменту выполнения.

- Значение переменной равно нулю, а вы пытаетесь разделить другое число на это значение.
- Вы пытаетесь взять данные из файла, который не открыли. Некоторые языки откроют его, многие выдают ошибку.
- Вы пытаетесь считать из массива длиной пять шестой элемент. Многие языки выдадут вам ошибку времени выполнения, так как такая операция указывает на ошибку программиста. Кроме того, отрицательные индексы массивов во многих языках ведут к ошибкам времени выполнения.
- Вы пытаетесь с помощью функции `parseInteger()` вместо строчной переменной `42` преобразовать в число переменную `суперсладкая кошечка`, так как считали не те данные. Обычно это влечет за собой ошибку времени выполнения. Некоторые языки считают, что числовое значение `суперсладкой кошечки` равно нулю, однако мы считаем, что больший смысл в данном случае имеет ошибка времени выполнения.

Логические ошибки — это ошибки времени выполнения, которые редко приводят к вынужденной остановке работы программы, но заставляют программу вести себя не так, как мы ожидаем. Часто они остаются «тихими», так как не приводят к появлению сообщения об ошибках, из-за чего их сложно заметить. Чтобы отследить их появление, вы должны с помощью команды `print` или отладчика (больше о них обоих читайте в дальнейшем) шаг за шагом отследить процессы обработки данных, происходящие в вашей программе.

Например, вызовите функцию с перепутанными параметрами. Несмотря на это, она все равно часто работает с данными и рассчитывает выходное значение — конечно же, неправильное. Счетчик цикла со значением на единицу меньше или больше нужного может приводить к тому, что вы считываете неправильные данные из массива.

Многие проблемы неохотно выходят из кустов, когда вы включаете предупреждения и сообщения об ошибках. Но когда-нибудь данная возможность исчерпает себя. Код синтаксически правилен и не содержит никаких очевидных для редактора или компилятора источников ошибок, но, несмотря на это, не работает. Речь идет о логических ошибках, и, чтобы их найти, нужно знать, что же, собственно, должна делать программа. Этого компьютер, конечно же, не может. В зависимости от языка, среды разработки, способностей и предпочтений программиста существует широкий набор стратегий, которые вы можете использовать, чтобы отыскать ошибку.

Инструменты и стратегии диагностики

Некоторые из описанных в данном разделе инструментов уже встроены в среду разработки (см. раздел «Среды разработки» главы 20). Некоторые языки, например Objective-C или Java, почти всегда предполагают программирование с использованием интегрированной среды разработки, но для других языков это скорее нехарактерно. Так как вопрос о том, какую среду разработки применять, является почти что религией, мы не даем никаких рекомендаций, но вам следует хотя бы один раз установить и опробовать среду, прежде чем вы от нее откажетесь.

Привычные подозреваемые

Есть несколько основных правил определения того, в каких местах вероятнее всего находится ошибка. Охотник пойдет туда, где он встретит дичь с наибольшей вероятностью, скорее на поляну в лесу, чем в собственный погреб с картофелем.

Правила отладки от Джо[1] говорят нам, что все ошибки находятся в диапазоне трех команд до и после последней измененной строки. Это означает, что после каждого изменения вы должны сразу же проверять их или запоминать, что вы изменили последним. Если не можете это запомнить, нужен контроль версий. Если у вас под рукой утилита сравнения (см. главу 20), то вы сможете очень быстро найти различия между двумя версиями. Очень вероятно, что одна из выделенных цветом строк и содержит ошибку.

Конечно же, мир не всегда настолько прост. Совершенно не исключено, что, хотя ошибки и присутствуют скрытно где-то в коде, они становятся заметны лишь тогда, когда вы что-либо измените в другом месте. Такие «спящие» ошибки сложно найти, так как, руководствуясь правилами от Джо, вы будете искать не в том месте. Утешает лишь то, что они относительно редки. Подавляющее большинство ошибок находятся в последних строках, которые вы изменяли.

Такими же подозрительными являются классы и части кода, в которых раньше уже возникали ошибки. Это связано с описанной ранее высокой вероятностью того, что при поиске ошибок вы лишь затушевываете, перемещаете или устраняете ошибку, но создаете при этом новую. Если ошибка ни там, ни здесь не бросается в глаза, вы можете ограничить подозрительные области, снова и снова удаляя участки кода или создавая комментарии.

Однако не сосредотачивайтесь слишком сильно на таких подозрительных местах. Предвзятое мнение о том, где находится ошибка, может (см. главу 12) вызвать то, что вы не заметите симптомы, свидетельствующие о другом диагнозе.

Валидаторы, Lint-программы, анализ кода

Если вы уже работали с HTML, CSS или родственными технологиями, то, вероятно, когда-то уже пользовались валидатором. Это программа, которая может найти и показать ошибку в исходном коде. Для настоящих языков программирования также существуют подобные инструменты, так называемые *Lint-программы* (от англ. lint — «пух»), так как представляют собой «щетку для очистки от пуха» для кода. В их названиях для каждого языка встречается корень lint, например: Pylint (для Python), JavaScriptlint или флаг компиляции -Xlint для Java. Они ищут подозрительные вещи, сделанные программистом, например использование неинициализированных переменных, и выдают список возможных ошибок. Они относятся к семейству *статистических анализаторов кода*. Среди них наряду с совсем маленькими помощниками, следящими только за орфографией, есть объемные инструменты для анализа кода и поиска ошибок. Они решают многие

---

[1] *Сейбел П.* Кодеры за работой. — М.: Символ-Плюс, 2011.

задачи из тех, что пришлось бы где-нибудь в другом месте решать компилятору, кроме процесса перевода в машинный код.

Более продвинутые находят также тип ошибок, который игнорируется компьютером, например, утечки памяти, так как они анализируют ход выполнения программы и устанавливают, что один блок кода выполняется всегда (файл открывается), а другой — лишь при определенных условиях (файл снова закрывается). Конечно же, все преимущества данных программ проявляются лишь в статически типизированных языках, таких как C, C#, C++ или Java, поскольку в этих языках типы переменных (см. раздел «Типы переменных» главы 26) очень жестко закреплены и не происходит динамического преобразования типов. Процесс компиляции в этих языках полностью отделен от старта программы, этим языкам неизвестны динамические конструкты, например `eval()`, компилирующие сгенерированный к моменту запуска программы исходный код. Поэтому статистический анализатор кода в этих скорее статических языках может гораздо лучше предсказать ход выполнения программы и распознать ошибки. Напротив, в динамическом языке всегда может произойти то, что отсутствующая функция появится лишь в ходе выполнения программы.

Если используемый вами язык поддерживает *строгий режим (strict mode)*, следует его включить. В таком случае языковой парсер или валидатор будет гораздо строже относиться к подозрительным участкам кода: то, что в обычном режиме было бы простым предупреждением, рассматривается как ошибка. Эта строгость помогает избегать скрытых ошибок.

Отладка с помощью команды print

> Боги программирования говорят: «Ты должен записывать команду print в места твоего кода, в которых что-то не так, а затем компилировать и запускать ее».
>
> *Джо Армстронг. Кодеры за работой*

Проверенным методом поиска ошибок является разбиение кода с помощью вывода журналов, которые подобны просекам для защиты от лесного пожара. В начале функции вы выводите журнал «Здесь начинается функция getBookmarks()», а в конце — «Здесь заканчивается функция getBookmarks()». Между обоими выводами журналов вы пишете другие команды, выводящие текущее содержание переменных, состояние соединений с базами данных, временные отрезки или перехваченные исключения. Если при первом выводе журналов все в порядке, а при втором — нет, ошибка в коде находится между ними. Часто вы будете сталкиваться с тем, что функция вызывает другие функции, в этом случае начините такими выводами журналов в начале и конце и другие функции, пока не будете уверены в том, что нашли ошибку в одной из функций между двумя такими просеками. В зависимости от длины функции вы можете искать ошибку прямо в коде или повторить процедуру с ограничением области еще раз.

Часто этот метод вкупе с использованием команды `print` — единственный возможный метод отладки, к сожалению, не очень эффективный, так как вам

приходится вписывать команды `print` в подозрительных местах по всему коду. Редко вы сразу же попадаете на нужное место, и тогда приходится много раз запускать программу, оценивать выводы журналов, вписывать новые команды `print` и запускать снова.

Оценка состояния программы с помощью журналов также называется *printf-отладкой*. Она использовалась еще в то время, когда C был самым популярным языком программирования, и получила свое название из-за функции `printf` (Print Formatted), использовавшейся в C. Пример:

```
for (i = 0; i < endPoint; i++) {
    printf("Loop: %d\n", i);
}
```

Значение счетчика цикла `i` при каждом запуске выдается один раз.

Специально для языков баз данных (во многих базах данных программы можно запускать прямо внутри базы) альтернативным вариантом является создание таблиц журналов и заполнение их значениями.

У отладки с помощью команды `print` есть один недостаток: вы очень быстро перестанете справляться с объемом информации, выводимой программой (порой это уже называют слепотой прокрутки). Если вы в десяти местах программы используете команду `print`, а затем, например, считаете данные из CSV-файла средних размеров, вам придется прочесать пару тысяч строк.

Ведение журналов

Ошибки времени выполнения, достаточно серьезные для того, чтобы привести к вынужденной остановке вашей программы, вы заметите сами. Однако есть ошибки, последствия которых не столь драматичны, но, несмотря на это, они должны быть зафиксированы: используете ли вы данные какого-либо сервера и желаете сохранить определенную ориентацию в ситуации, как часто соединение с данным сервером невозможно, либо у вас есть функция, в которой могут появиться исключения, и вы хотели бы каждый раз вносить запись в журнал, если это происходит. Возможно, вы также хотели бы в определенных местах вашей программы заносить в журнал значения переменных, чтобы потом проследить, правильно ли ваша программа их рассчитала.

Вы можете создавать журналы с помощью команды `print`, но лучшим решением будет использование библиотеки журналов. Во-первых, такие библиотеки стандартизованы, во-вторых, вы можете установить в настройках, сколько и какие диагностические сообщения должны выводиться. Типично наличие журналов разных уровней, от `Fatal` (система больше не работает, кто-то должен проверить ее и исправить ошибку) до `Debug` (вся системная информация, которую можно вывести, будет сохраняться в журнал). Как называются промежуточные ступени и сколько их, зависит от библиотеки: чаще всего есть `Info` (довольно большое количество выводов журналов; правильный параметр, чтобы впервые протестировать программу на новом компьютере), `Warning` (выводятся лишь проблемы, но такие, которые не мешают работе программы) и `Error` (серьезные ошибки, которые могут сказаться на ходе выполнения программы).

При использовании библиотеки журналов вы в зависимости от случая решаете, какой уровень журналов хотели бы присвоить информации, вносимой в журнал. Например, если вы перехватили исключение, то можете внести эту серьезную ошибку в журнал:

```
Logger.log(LogLevel.ERROR, "HostedMode exception: ", e);
```

Значение переменной из той же самой программы будут вноситься в журнал с уровнем `Debug`:

```
Logger.log(LogLevel.DEBUG, "Users connected:"+numConnectedUsers);
```

При запуске программы вы можете внести в журнал сведения о конфигурации программы, чтобы потом можно было легко проверить, произошел ли запуск системы с правильными настройками:

```
Logger.log(LogLevel.INFO, "Configured to use database instance
"+Config.databaseName);
```

Можете задать в настройках библиотеки журналов, будут ли записываться в библиотеку все журналы или только важные. Если вы занимаетесь отладкой программы, то `Debug` или `Info` — хороший выбор, тогда в ваш файл журналов действительно будут записываться все команды журналов. Если вы столкнулись с тем, что тонете в потоке сообщений, выставьте уровень журналов с меньшим количеством уведомлений, например `Warning`. Таким образом, сообщения с уровней `Debug` или `Info` исчезнут из файла журналов без каких-либо изменений кода с вашей стороны. Для теста ПО в большинстве случаев подходят `Warning` и `Error`, и если программа работает хорошо, установите в настройках `Error` или `Fatal`, чтобы важные ошибки не затерялись в потоке несущественных отладочных сообщений.

Если вы используете язык, в котором нет возможности работать с уровнями журналов, и вы не хотели бы использовать фреймы журналов, вам следует как минимум найти возможность настраивать ваши журналы. Чтобы убрать выводы команды `print` одной функции, вы можете задать локальную булеву переменную `debug` и сделать команды `print` зависимыми от ее состояния:

```
function readAnalysisData (fileName) {
    var debug = true;
    ...
    if (debug) {printf("readAnalysisData, checkpoint1,\n", i);}
}
```

Если вы зададите ей значение `false`, выводы журналов данной функции исчезнут.

Если вы работаете на базе Unix (Mac или Linux) или используете Cygwin в Windows, можете записать все команды `print` в один документ, например, запустив программу с `./myprogram >log.txt` и используя `tail -f log.txt`, чтобы просмотреть вывод (также см. главу 22).

Вы также можете записать ваши данные, разделив их запятыми или знаками табуляции, и после завершения работы программы проанализировать их в Excel или другом табличном процессоре. Если вы используете команду `print` много раз,

вписывайте в отладочный вывод в качестве маркеров различные короткие строчные переменные:

```
function readAnalysisData (fileName) {
    printf("readAnalysisData, enter,\n", i);
    file = open(fileName);
    for (i = 0; i < file.length; i++) {
        printf("readAnalysisData, loopstart, %d\n", i);
        ...
        printf("readAnalysisData, loopend, %d\n", i);
    }
    printf("readAnalysisData, exit,\n", i);
}
```

Запустите программу с `./myprogram >log.csv`, и вы получите CSV-файл, выглядящий примерно так:

```
readAnalysisData, enter,
readAnalysisData, loopstart, 0
readAnalysisData, loopend, 0
readAnalysisData, loopstart, 1
readAnalysisData, loopend, 1
readAnalysisData, exit
```

Затем, если вы импортируете этот файл в табличный процессор, можете сортировать по столбцам, например, чтобы убедиться, что и `loopstart`, и `loopend` были внесены в журнал. Если вы хорошо разбираетесь в своем табличном процессоре, то можете с помощью фильтра столбцов целенаправленно посмотреть лишь данные по функции `readAnalysisData`. Это занимает много времени, однако иногда может сэкономить время, которое вы в противном случае провели бы, угадывая, в чем проблема.

Если ваша программа обрабатывает большие массивы данных, создайте тестовый массив, который намного меньше. Соответственно, программа будет создавать гораздо меньше журналов. Если определенная ошибка не появляется в маленьком тестовом массиве, вы должны насторожиться: возможно, в оставшейся части большого массива скрывается ошибка в данных.

Отладчики

Отладчики — вспомогательные инструменты, наблюдающие за работающей программой. Используя их, вы можете проверить значения переменных в момент их изменения и следовать за ходом выполнения программы с помощью различных файлов, объектов и/или функций. Благодаря отладчику вы можете точно отследить, что на самом деле делает программа, вместо того чтобы лишь предполагать. Отладчик может помочь вам разобраться и в коде без ошибок, понять работу которого вам сложно, так как в этом случае вы можете проверять свои предположения сразу же.

Магическое действие отладчика состоит в том, что он может «прикрепиться» к работающей программе и в заданных вами пунктах (так называемых *точках останова*)

остановить ее. Когда выполнение программы останавливается в такой точке, вы можете сделать следующее.

- Считать все видимые на данный момент значения переменных и даже изменить их.
- Продолжить или прекратить работу программы с этой точки.
- Наиболее интересная особенность состоит в том, что вы можете разбить ход выполнения программы на шаги: исходный код будет обрабатываться строчка за строчкой и после каждой строки отладчик будет останавливать программу, чтобы вы могли посмотреть значения переменных.

Для большинства распространенных языков программирования существуют отладчики, работающие на базе исходного кода: неважно, программируете ли вы на скриптовом языке, в JavaScript, в браузере или в компилируемом языке, в реальности ваш исходный код обрабатывается не процессором, а более или менее оптимизированной программой обработки машинного кода, который создан из исходного кода. Вам не нужно программировать на языке программирования, а отлаживать на машинном языке, поскольку отладчики настолько продвинуты, что лживо уверяют вас, что вы можете шаг за шагом двигаться по своему коду и наблюдать за работой прописанных вами вычислительных операций. Этот метод, получивший названий Sourcemapping, является маленьким техническим чудом, которое делает отладчик гораздо более удобным для пользователя. На рис. 13.1 вы видите расширение Firebug для Firefox, которое очень популярно у разработчиков интернет-страниц.

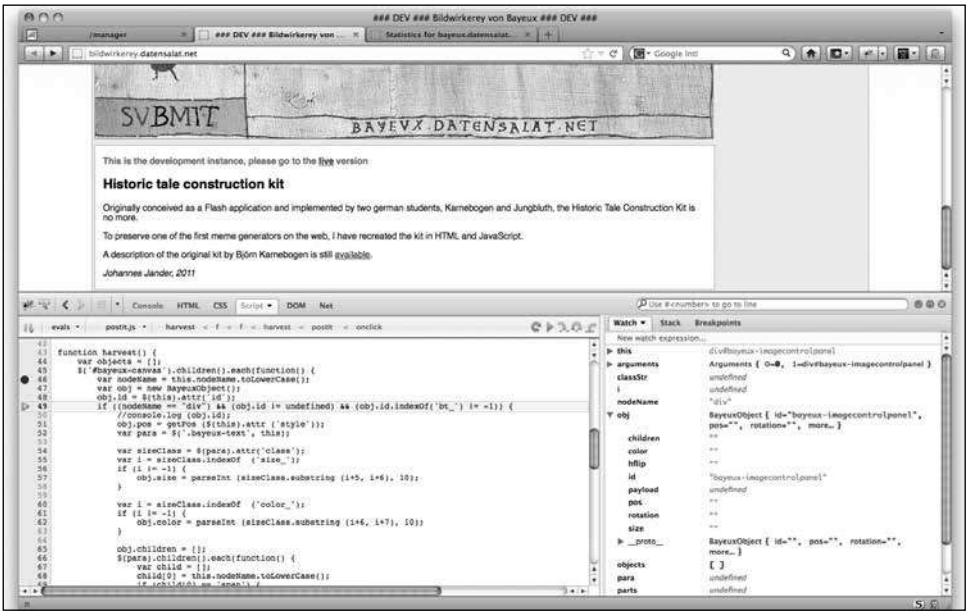


Рис. 13.1. Firebug в качестве среды разработки — JavaScript Sourcelevel Debugger

Существует две версии отладчиков: запускаемые через командную строку (они прилагаются к большинству языков программирования) и интегрированные в среду разработки. Принципы их работы в основе своей одинаковы, но сфера использования и задачи сильно различаются.

То, как работает отладчик, мы объясним на примере маленькой программы, написанной на Ruby, — один раз для интегрированного отладчика, второй раз — для отладчика, запускаемого из командной строки. Программа считывает TSV-файл и для каждой строки подсчитывает графы. Количество граф выводится вместе с их значениями. Для этого программа использует команду `count_terms`, которая вызывается циклом в каждой строке (функция определена вверху, основная функция начинается ниже):

TSV (TAB-SEPARATED VALUES)

Под TSV понимается возможность представления таблиц в формате «только текст». Данные в отдельных графах таблицы разделены знаками табуляции.

```
#!/usr/bin/env ruby

def count_terms(terms)
    return terms.length;
end

File.open("testdata.tsv", "r").map{|line|
    terms = line.split("\t");
    len = count_terms(terms);
    puts "#{len} terms in #{terms.join(' ')}";
}
```

Отладка в интегрированном отладчике

При загрузке программы в интегрированную среду разработки мы можем щелчком на панели внизу слева установить точку останова, которая на рис. 13.2 появляется на той же панели в виде кружка. Указатель там же показывает нам, в какой строке только что была остановлена программа.

У интегрированного отладчика есть ряд полей для вывода информации и полей управления, на рис. 13.2 в нижнем поле read_tsv.rb представлен исходный код программы. Его выполнение было остановлено на девятой строке. Программа считала `tsv`-файл и разделила строки, сейчас переменной `terms` будет присвоено значение первой строки в качестве массива, отделенного знаками табуляции. Вверху справа, в поле с переменными, этот массив разложился и мы видим его первые элементы. Верху слева, в поле Debug, нам показан ход выполнения программы и то, что программа была остановлена. Текущий вывод программы после стандартного вывода данных отображается в поле Console.

Красный прямоугольник в поле Debug *(2)* прерывает программу, зеленая стрелка (run) *(1)* продолжает работу программы до следующей точки останова, желтые указатели *(3–5)* разбивают последующий ход выполнения программы на маленькие шаги.

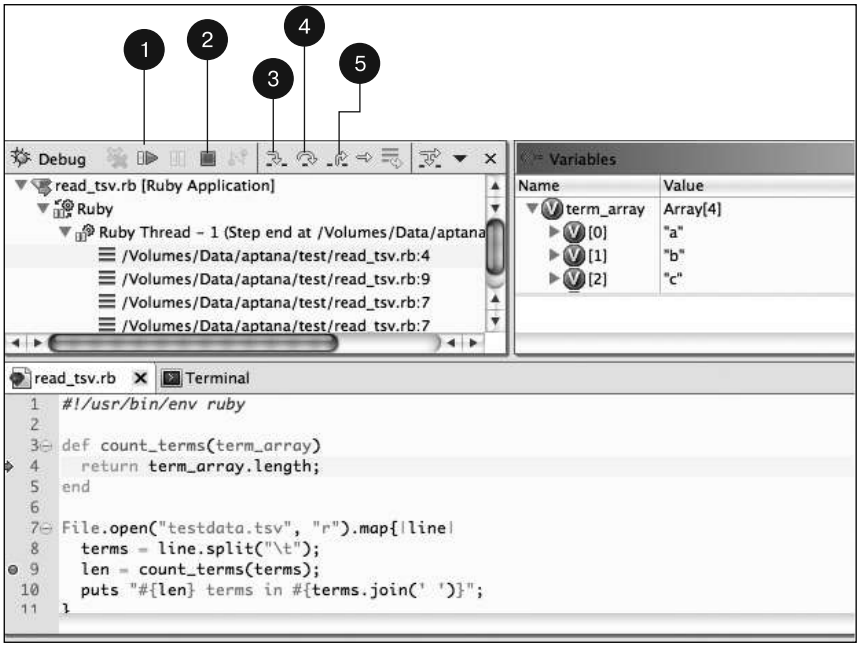


Рис. 13.2. Интегрированный отладчик — программа была приостановлена

Отладка в отладчике, запускаемом через командную строку

Команды данного типа отладчика приблизительно совпадают с командами встроенного отладчика:

- установить/удалить точку останова;
- продолжить работу до следующей точки останова (run);
- переместиться в функцию и остановить в ней программу (Step into);
- переместиться из текущей функции и остановить программу (Step out);
- выполнить одну строку и остановить программу (Step over).

Для запуска скрипта в отладчике Ruby наберите:

```
ruby -rdebug myscript.rb
```

Скрипт запустится и остановится на первой строке. Затем от вас потребуется ввести одну из следующих команд.

- `b(reakpoint) <номер строки >` — устанавливает точку останова в данной строке.
- `n(ext)` — продолжает выполнение скрипта и останавливается в следующей строке (наша команда Step over, см. ранее).
- `s(tep)` — выполняет функцию в текущей строке и останавливается в ее начале (наша команда Step into, см. ранее).
- `fin(ish)` — продолжает выполнение скрипта и останавливается после завершения текущей функции (наша команда Step out, см. ранее).

- `c(ontinue)` — скрипт выполняется до следующей точки останова (run).
- `c <номер строки>` — скрипт выполняется до указанной строки и останавливается в ней.
- `p(uts)` — выводит значения переменных.

В отладчиках есть три команды для пошагового выполнения кода: Step into, Step over и Step out.

Пошаговое выполнение в интегрированном отладчике

Стрелка в середине (см. рис. 13.2, *4*) обозначает Step over. Щелчок на ней приводит к тому, что функция `count_terms()` выполняется, но программа снова останавливается только после этого. Тогда вы попадаете на экран, показанный на рис. 13.3. Вам как пользователю кажется, что программа пропустила эту функцию, однако у переменных, изменяемых этой функцией, после этого изменились значения. Step over — стандартная команда для того, чтобы каждый раз не проверять все функции со всеми их подфункциями.

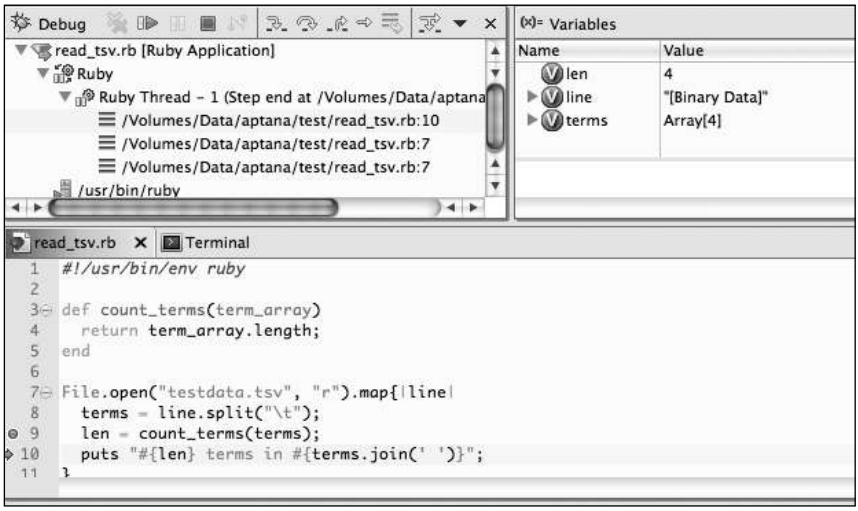


Рис. 13.3. Интегрированный отладчик после Step Over

Стрелка слева (см. рис. 13.2, *3*) обозначает Step into. Это означает «выполни функцию в этой строке и остановись». Если вы переходите в какую-либо функцию, то видите локальные переменные с их значениями. Можете перейти еще дальше, в подфункции, и с помощью команды Step over просмотреть шаг за шагом все операции выбранной функции.

В данном случае мы снова запустим программу в отладчике до точки останова и перейдем в функцию `count_terms()`. В тот момент, когда программа дошла до выполнения функции в точке останова, мы щелкаем на стрелке Step into и попадаем в окно, показанное на рис. 13.4.

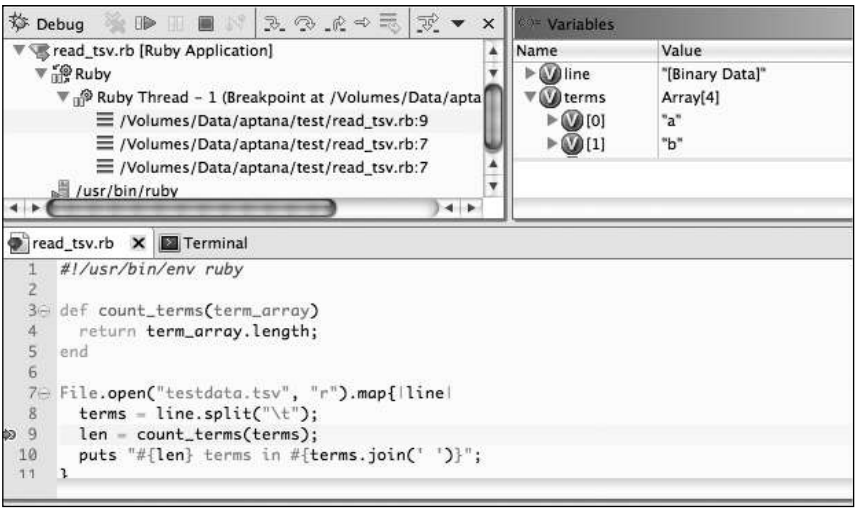


Рис. 13.4. Интегрированный отладчик после Step Into

Кружок точки останова по-прежнему стоит напротив девятой строки, однако указатель — напротив четвертой. Программа находится сейчас в функции `cnt_terms()` и ждет дальнейших указаний. Сверху справа, в поле Variables, вы видите локальные переменные функции, поэтому здесь показывается `term_array`, а не `terms`. Всегда, когда у вас есть подозрение, что в функции может быть ошибка, вы должны с помощью Step into перейти в эту функцию и проверить расчеты на основе значений переменных.

Стрелка справа (см. рис. 13.2, *5*) обозначает Step out. Щелчок на ней ведет к тому, что программа выполняет остаток текущей функции до конца и затем останавливается. Итак, если вы щелкаете где-либо в функции на `count_terms()`, то попадаете в окно, изображенное на рис. 13.3. Все расчеты в функции были произведены, и, как и должно быть, использование функции закончено. Step out применяется после того, как вы с помощью Step into перешли в функцию и убедились, что она все делает верно, — с помощью Step out вам не нужно выполнять функцию до конца, и вы можете продолжить уровнем выше.

Типичный прием при работе в отладчике — установка точек останова перед предполагаемым местом с ошибкой, остановка программы в этом месте и проверка того, имеют ли смысл значения переменных. Что это означает в отдельно взятом случае, зависит от типа ошибки, которую вы хотите устранить. Если речь идет о делении на ноль, вы устанавливаете точку останова там, где рассчитывается делитель, то есть значение в знаменателе. В случае исключения вы, как правило, можете считать номер строки, в которой оно появилось, в журнале и для пробы установить точку останова за пару строчек до этого. Затем вы шаг за шагом продвигаетесь к подозрительной строке и наблюдаете за тем, какие функции и значения переменных задействованы. Часто затем вам приходится прерывать ход программы, устанавливать новую точку останова и запускать программу еще раз.

Пошаговое выполнение в отладчике, запускаемом из командной строки

Запустим нашу программу-пример не в интегрированном отладчике, а в отладчике командной строки в Ruby, используя `rdebug`:

```
johannes$ ruby -rdebug read_tsv.rb testdata.tsv
read_tsv.rb:3:def count_terms(terms)
```

Программа запустилась в отладчике, однако остановилась уже в первой функции `count_terms(terms)`:

```
(rdb:1) b 9
Set breakpoint 1 at read_tsv.rb:9
```

Мы устанавливаем точку останова в строке 9, в которой стоит `len = count_terms(terms);`:

```
(rdb:1) c Breakpoint 1, toplevel at read_tsv.rb:9
read_tsv.rb:9: len = count_terms(terms);
```

Программа выполняется дальше и останавливается в точке останова в строке 9. Отладчик дружелюбно показывает нам, какой код стоит в этой строке:

```
(rdb:1) s read_tsv.rb:4: return terms.length;
```

Переходим в функцию `count_terms(terms)`. В ней отладчик останавливает программу и снова выводит нам исходный код:

```
(rdb:1) p terms.length
4
```

Мы можем вывести в отладчике длину массива `terms` — 4:

```
(rdb:1) s read_tsv.rb:10: puts "#{len} terms in #{terms.join(' ')}";
```

Идем дальше и покидаем функцию `count_terms(terms)`:

```
(rdb:1) s 4 terms in a b c d read_tsv.rb:8: terms = line.split("\t");
```

Затем идем дальше и видим результат выполнения команды `puts (print)`.

Отладка в декларативных языках программирования

При программировании с помощью декларативных языков (см. раздел «Языковые семьи» главы 26), таких как SQL или CSS, программист не задает пошаговое руководство того, как программа должна решить текущую задачу, а с помощью селекторов и правил определяет, какие элементы должны изменяться. Компьютер обладает относительно высокой степенью свободы в том, как он их будет использовать.

С одной стороны, это преимущество, поскольку, как говорит опыт, в декларативных языках появляется меньше серьезных ошибок. Нам гораздо легче представить себе «я хочу это», чем держать в голове и корректно записать во всех

деталях 100 шагов, которые ведут к цели. Но, с другой стороны, при появлении ошибок, если, например, оператор SQL не выдает результат, хотя данные в таблицах есть, или регулярные выражения выдают только пустые строчные переменные, или команды CSS не работают, ситуация становится щекотливой. В декларативных языках нет возможности воспользоваться способом `print`-отладки, чтобы во время выполнения программы вывести значения переменных. Маленькими шажками двигаться по коду и наблюдать за тем, когда какие значения изменяются, также невозможно.

Но не все так безнадежно, так как и в декларативных языках есть возможности заглянуть внутрь системы, о которых мы поговорим в следующих разделах.

SQL

`explain` — мощный инструмент, если ваш SQL очень медленный, так как `explain` точно говорит вам, как база данных обработала запрос. Приобретя некоторый опыт, вы сможете заметить, где базе данных пришлось просмотреть весь столбец в таблице, так как вы не определили индекс. При отладке запросов, которые либо не выдают строки, либо выдают их слишком много, это, к сожалению, помогает очень слабо. Такие запросы чаще всего задаются командой `Join`, которая связывает много таблиц, и если в них присутствует пара нулевых значений, программа не выводит никаких значений. Но если условия для данного оператора слишком отличаются от стандартных, что ведет к тому, что программа выдает много строк, `explain` может указать вам на место возникновения проблемы, показывая величину массивов данных на разных уровнях сложного запроса.

Здесь вам помогут разложение запроса на компоненты и их проверка по отдельности. Создайте тестовый массив данных, высчитайте в уме и на бумаге правильный результат и камень за камнем снова соберите ошибочный запрос. В какой-то момент совокупность результатов начнет отклоняться от ваших ожиданий, тогда следует ввести что-нибудь вроде `left join` или других специальных условий.

Реляционные системы баз данных, понимающие команду `WITH`, облегчают эту работу. `WITH` позволяет выделить различные подзапросы в отдельные блоки, которые визуально отделены от запросов с `Join`. Таким образом, вы легко можете скопировать и проверить любой из этих подзапросов. Если все подзапросы работают, но общий результат все-таки еще неверен, проблема в главном запросе.

Регулярные выражения

Регулярные выражения сложны для чтения, понимания и, к сожалению, также для отладки. Так как в них в сжатом виде закодированы действительно сложные операции, нам тяжело безошибочно проследить за ними. То, что в регулярных выражениях производится группировка и одна из групп часто является опциональной, не делает более легким понимание и поиск ошибок.

Несколько лет назад появились вспомогательные инструменты, с помощью которых можно регулярные выражения интерактивно разложить и отладить. Один из примеров — `debuggex.com`, страница, на которую вы вносите ваши регулярные

выражения и тестовую строчную переменную (рис. 13.5). Запрос в поиске `regex tester` выдаст еще предложения. Приложение показывает вам, производится ли группировка, какие группы связываются (номера строк в скобках, которые вы задаете, чтобы использовать результат как регулярное выражение), и создает пару случайных, подходящих под ваши регулярные выражения строчных переменных.

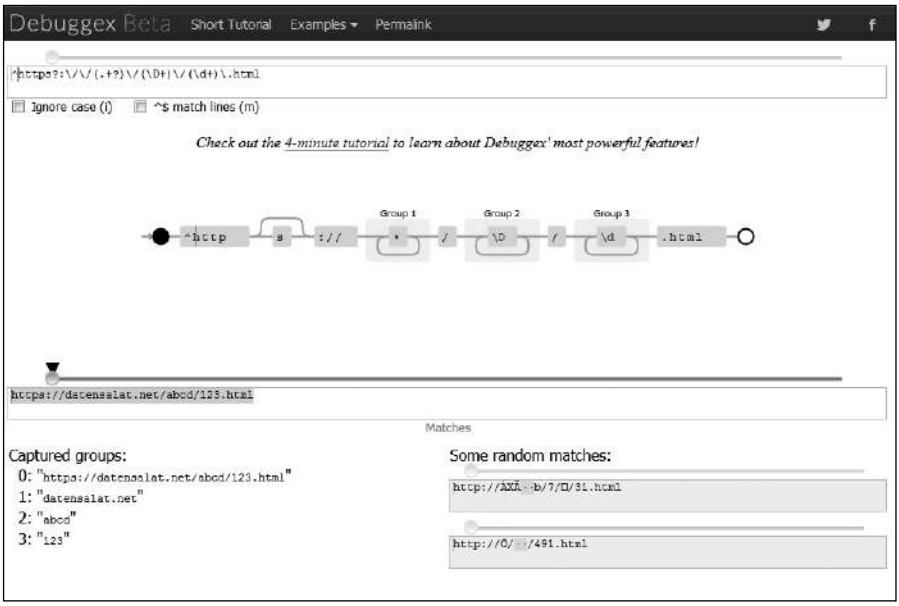


Рис. 13.5. Debuggex: вид регулярного выражения

RegexBuddy — приложение Windows, которое выдает еще более подробную информацию в цвете о результатах, поиске с возвратом[1] и шагах, ведущих к результату.

CSS

Во все браузеры встроены инструменты разработчика. Эти инструменты выдают красивый список CSS-команд, действующих для выделенного фрагмента HTML. CSS-команды, затертые другими, будут отображаться перечеркнутыми.

Если одна из ваших команд с указанием стиля отсутствует, возможно, вы сделали опечатку, так как синтаксические ошибки в CSS ведут к тому, что команды вообще не вносятся в список. Или же вы забыли сохранить ее, или сохранили не в тот файл — в любом случае отсутствие команды будет тем, что вам поможет. С помощью CSS-валидаторов, например CSS Lint (`csslint.net/`) или Jigsaw (CSS-валидатор от W3C) (`jigsaw.w3.org/css-validator/`), вы сможете быстро найти синтаксические ошибки. Почему команды затираются другими (например, в другой команде

[1] Одна часть регулярных выражений не сработала, поэтому выполнение продолжается с другой частью.

стоит специальный селектор или же она находится в связанном позднее файле), эти инструменты, к сожалению, сказать не могут, однако они как минимум дают вам отправную точку для понимания того, почему эти команды не работают.

Если в разных современных браузерах вы видите большие различия, предположительно, в вашем CSS или HTML есть ошибка, так как на данный момент все браузеры выдают одинаковые результаты, вплоть до мелких деталей. Или, возможно, причина в том, что вы пытаетесь с помощью лома связать два CSS-кода, несовместимых друг с другом. Строчные элементы не любят команды `height` и `width`, они предназначены для элементов блочного уровня. И наоборот, `line-height` — только для строчных элементов.

Если элементы упорно ведут себя не так, как вы хотите, и вы проверили CSS и HTML на предмет ошибок, уменьшите количество команд стиля, в особенности тех, которые влияют на тип отображения и положение на странице, так как они могут вступать в хитрое взаимодействие. Например, включите в код `float: left`, `position: absolute`, а также `display: block`, однако команда `float` без указания ширины не растягивает объекты по ширине на 100 %, а сжимает ваше содержимое, то есть работает скорее как `display: inline-block`.

Если больше ничего не помогает

Если вас затерло на какой-то проблеме, которая кажется неразрешимой, часто помогает физически или хотя бы мысленно отойти от клавиатуры. Прогулки, плавание и поездки на велосипеде часто вызывают хорошие идеи, так как при этом через наш мозг медленно прокатываются альфа-волны, способствующие появлению творческих решений. Часть мозга занимается моторной деятельностью, внимание блуждает где-то в другом месте, и именно поэтому порой можно как будто краем глаза воспринять некоторые аспекты проблемы, которые при полной концентрации остались скрытыми. Кроме того, крайне полезно то, что в движении вас не может отвлечь Facebook.

Во время сна в нашем мозгу появляются более медленные тета-волны, которые могут вытащить на свет искомый результат. Сон, как выяснили исследователи, и сам по себе помогает в решении проблем. Иногда решение само собой приходит на следующее утро. Вместо того чтобы за ночь сломать себе все зубы, грызя задание, можно просто пойти спать. Временами решение, найденное во сне, относится к проблеме, о существовании которой вы и не подозревали.

> Мне снился ужасный пробел в безопасности в огромном механизме Blogengine. Когда я проснулась, я поняла, что этот пробел действительно существует. Каждый шутник мог заменить все изображения в блоге картинкой со своей отсканированной пятой точкой.
>
> *Катрин*

Конечно же, кроме места для сна, вам нужен лист бумаги или что-нибудь другое для записи новой идеи. Иначе на следующее утро она исчезнет — неважно, насколько настойчивой и незабываемой она казалась вечером. И кто знает, придет ли она когда-нибудь снова.

Сон на рабочем месте — не всегда доступный вариант, особенно для служащих. Небольшое утешение: усталость тоже может помочь при решении проблемы. Если задать испытуемым задачи, требующие творческого решения, лучше всего они с ними справляются в те часы, когда наименее бодры, например, совы — по утрам и наоборот. Пьяным студентам также удавалось найти неожиданные решения быстрее, чем трезвым. Уставший и плохо соображающий мозг хуже сопротивляется различным странным идеям и неожиданным связям, и именно эти идеи являются тем, что нужно при решении определенных проблем[1].

Одной из самых простых и излюбленных стратегий поиска ошибок является метод утенка. Своим названием метод обязан неизвестному программисту, у которого была привычка с целью поиска ошибок объяснять свой код утенку, стоящему на рабочем столе. Если у вас нет под рукой утенка, можете воспользоваться помощью любого человека, стоящего рядом, от которого не требуется разбираться в программировании лучше, чем пластмассовая игрушка. Строка за строкой опишите этому человеку, что должен делать код и что он делает вместо этого. Во многих случаях очень быстро вы скажете: «О! Это же там не написано. Спасибо за помощь, теперь я знаю, в чем дело». Метод утенка может быть полезным уже на стадии планирования: когда вы описываете другому свои намерения, путь их исполнения становится более понятным. То, что вы облекаете свой план в слова, явно задействует иные области мозга, нежели его обдумывание.

> Раньше я использовала для этого разные вещи на моем письменном столе, потом резинового утенка (который потерялся) и в конце концов фигурки из «Мой маленький пони». Игра в вопросы и ответы с пони в зависимости от типа фигурки пони из сериала иногда помогала мне в решении проблемы. Недавно я завела кошку. Теперь я говорю с ней, если мне нужно обсудить с кем-то проблему.
>
> *www.flickr.com/photos/chix0r/8419684010*

Для среды разработки IntelliJ-Idea существует Code Consultant Plugin (sites.google.com/site/codeconsultantplugin). Code Consultant задает вопросы, относящиеся к делу, например: «Это правильный класс?» или «И о чем нам это говорит?». Однако разработчик, Натан Вокленд, предупреждает: «Плагин создан лишь для разговора. Если вам действительно нужна помощь, поговорите с коллегой или другим человеком». Если поблизости нет ни коллеги, ни плагина, ни утенка, вас в любое время выручит Developer Duck: www.developerduck.com.

Родственным методу утенка является метод отладки Stack Overflow, название которого мы взяли из комментария к Hacker News:

> Я называю это методом Hacker News. Я уже решил много проблем, пытаясь сформулировать хороший вопрос для Stack Overflow. Проблема кажется сложной, если я формулирую ее в уме одним предложением. Но если я пытаюсь описать, что скрывается в глубине вопроса, чего я хочу

---

[1] Более подробную информацию можно найти на сайте www.wired.com/wiredscience/2012/02/why-being-sleepy-and-drunk-are-great-for-creativity/.

добиться, какие инструменты использую, когда появляется моя проблема, и все это на примере упрощенных случаев, то чаще всего ответ приходит в голову еще до того, как я написал вопрос до конца.

news.ycombinator.com/item?id=5239925

Если и это не помогает

И от профессиональных программистов снова и снова можно услышать или прочитать, что они часто не знают, как именно им удалось устранить какую-либо ошибку. Вот что сказал один из программистов проекта Arpanet Берни Козелл: «Вот так я и заработал свою репутацию, исправляя ошибки, которые больше никто не мог исправить. Хорошо, что меня никогда не спрашивали, как я это делал. Потому что честный ответ, как правило, был: "Я недостаточно разобрался в коде, поэтому просто взял и написал новый"»[1]. Конечно, поясняет Козелл, он был очень хорош в написании нового кода — плохим программистам не стоит следовать его примеру.

Написать код заново — не самое быстрое или простое решение, но в жизни программиста в определенный момент возникает ситуация, когда код ни в какую не поддается отладке. Если вы подошли к этой точке, можно либо попросить совета у более опытного коллеги, либо переписать часть кода с ошибкой заново, лучше всего, не заглядывая в нерабочий код. Запишите для себя входные значения и задачи кода, а затем создайте, если получится, даже более хорошую версию. В итоге вы сэкономите время и нервы.

После поиска ошибок о поиске ошибок

Если вы обнаружили ошибку, то не думайте, что это определенно последняя в вашей жизни ошибка в программировании. Подумайте пару мгновений над следующим.

- Привела ли к этой ошибке одна из моих привычек программиста? Если да, то какая?
- Привел ли к ошибке концепт моего кода? Если да, то как?
- Как я мог бы быстрее напасть на след этой ошибки?

Если вы уже знаете, что забывчивы, запишите плод своих размышлений в тетрадь «выученных уроков». Пару минут, которые займут у вас раздумья и запись, вы сэкономите при поиске следующей ошибки.

Лучше всего — для каждой ошибки, которую вы устранили, записывайте механизм проверки. Так вы напомните себе об этом позднее, если ошибка появится вновь в случае, когда либо вы устраните не саму ошибку, а лишь ее симптомы, либо по каким-то причинам старый код снова «возродится», например, если вы удалили из системы управления версиями старый вариант. Как создавать правильные тесты для проверки, подробно рассказывается в главе 16.

---

[1] *Сейбел П.* Кодеры за работой. — М.: Символ-Плюс, 2011. — С. 457.

Наиболее распространенные причины ошибок плохих программистов

1. **Измененный файл не используется.** Если сделанные вами изменения ни к чему не приводят, ради проверки внесите абсурдное значение для какой-либо переменной, которое в любом случае будет заметным в работе кода. Если же и после этого нет никакой реакции, предположительно, вы изменяете не тот файл или же вносите изменения не в том месте. Возможно, путь в файле конфигурации или программе указан неправильно, что приводит к тому, что вы вносите правки в правильный файл, но программа выполняет неправильный.
2. **Переменная не инициализирована.** (Инициализация переменной выглядит как несколько излишняя строка, в которой `foo` установлено на 0 или указан пустой массив.) Эту ошибку часто допускают программисты в начале карьеры, сопровождая ее мыслью: «Пф, мелочные придирки! Работает же, как только что выяснилось, отлично и без этой бюрократии с инициализацией!» Во-первых, действительно работает, но затем программист вставляет свой код в цикл, может быть даже не зная об этом. Поэтому сейчас переменная вместо ничего все время получает остаточные значения из прошлого цикла и создает путаницу.
3. **Порядок оценки в длинных расчетах неправилен.** Выход из ситуации — дополнительные скобки, в том числе в тех местах, где они кажутся ненужными. Не обращайте внимания на насмешки других программистов над тем, что вы расставляете ненужные скобки. Это не стоит никаких усилий и, кроме того, закрепляет то, что вы действительно хотели сделать.
4. **Вы используете неправильную версию библиотеки, фреймворка или другого связанного инструмента.** В самом удачном случае не удается процесс компиляции, так как компилятор не может ничего сделать, используя данную версию. Это может случиться лишь в языках, компиляция в которых происходит до запуска программ (например, C++). Скриптовые языки, такие как PHP или JavaScript, подключают библиотеки по-другому, в них только ко времени запуска вы заметите, что используемая версия библиотеки несовместима с версией языка.
5. **Проблемы с регулярными выражениями, в особенности с greedy.** Вместо маленького кусочка регулярное выражение сразу бросается на половину мира. В таких случаях помогает `regex-tester` (см. ранее).
6. **Разным функциям по ошибке дали одно название.** Вы написали две функции с одинаковым названием и изменили настройки вывода в программе так, чтобы она никогда не выдавала соответствующее сообщение об ошибке. Небольшое утешение: все равно это обозначает достойную похвалы привычку последовательного присвоения имен функциям.
7. **Вызванная функция не та, которая имелась в виду.** Она лишь называется примерно так же. Это происходит прежде всего тогда, когда вы используете редактор, который автоматически дополняет названия функций и переменных, а программист

дает слишком похожие названия, к которым затем добавляются неправильные окончания.

8. **Всеобщая путаница с isset, empty, NULL и прочими им подобными** (см. раздел «Тяжелая работа с ничем» в главе 17).
9. **Ошибки неучтенной единицы.** При записи или чтении массива вы забываете, с чего начинается стартовый индекс — с нуля или единицы или как считается длина массива — с нуля или единицы, или исправляете программу не в том направлении. Дьявольским образом большинство языков начинают считать массив с нуля, но длина массива задается, начиная с единицы. Поэтому у вас возникают такие же неприятности, как, например, здесь:

```
var values = String[5]; // values сейчас ['','','','','']
var index = values.length(); // index сейчас 5
var val = values[index]; // values имеет значения для values[0]
                         // до values[4],
                         // однако функция обращается к values[5]
```

В категорию ошибок неучтенной единицы входят также частые ошибки, когда цикл выполняется слишком много или слишком мало раз. Виновником в большинстве случаев является `i = 0`, `i < max_value`, `i++`, в середине которого вместо `<` должно стоять `<=`.

14 Недобрые предзнаменования, или Коричневые M&M's

Код уже сейчас очень плохой.
— Это же дело вкуса.
— Ну да, но внутри есть операторы безусловного перехода.
— О...

Катрин Пассиг в разговоре с разработчиком ПО Аной Шюслер

Музыкальная группа Van Halen во всех контрактах требовала от организаторов, чтобы в служебных помещениях за сценой стояла тарелочка с M&M's, в которой не должно быть драже коричневого цвета. Звучит как легенда, но это не так. Дэвид Ли Рот обосновывает это требование в своей автобиографии.

> Van Halen была первой группой, которая устраивала огромные шоу на сцене в городах средней величины. Мы приезжали с девятью фурами оборудования туда, где до этого стандартом были максимум три. И технические проблемы не заканчивались никогда — подпорки сцены не выдерживали вес, пол проваливался, двери были недостаточно большими для нашего оборудования.
>
> Список наших требований был похож на китайский телефонный справочник, так как у нас было так много оборудования и людей, за него отвечающих. [...] Поэтому мы в качестве небольшой проверки в конце раздела с описанием технических требований поместили следующее: «В помещениях за сценой не должно быть никаких M&M's коричневого цвета; при нарушении этого правила группа имеет право отказать в предоставлении своих услуг с сохранением права на гонорар, закрепленный в договоре». Если же я находил в тарелочке коричневые M&M's, все нужно было основательно проверять сначала. Стопроцентно где-то скрывалась техническая ошибка. Они не прочитали договор. На них нельзя было полагаться. Иногда я находил при этом вещи, которые испортили бы все шоу. Находились в буквальном смысле опасные для жизни проблемы.
>
> *Дэвид Ли Рот. Одуревшие от жары*

Коричневые M&M's есть и в коде. Определенные сами по себе безобидные предзнаменования всегда указывают на то, что в фундаменте кода что-то не так. Этим объясняется резкая реакция профессионалов на мелочи вроде неунифицированного форматирования, которую новички воспринимают как преувеличение и придирки. Признаки, рассмотренные в следующих разделах, в случае удачи помогут вам увидеть и устранить такие сигналы в вашем коде. Даже если вы ничего не будете делать дальше, после прочтения этой главы вы хотя бы будете знать, как опытные программисты с одного беглого взгляда замечают, что ваш код скоро сорвет все шоу.

Слишком большие файлы

Начинающие программисты склонны записывать все в один большой файл, что делает код непрозрачным. Но вам ничем не поможет и тщательная разбивка кода на 12 файлов, если каждый из них сам по себе очень большой. Когда можно считать, что файл большой? В качестве основного правила достаточно знать, что пять экранных страниц — это много. Файл, содержащий десять важных функций (не маленьких помощников, которые лишь вызываются из этого файла), тоже довольно большой.

Проблема

Большой файл вызывает искушение небрежно разбить код. Новые функции просто «пришиваются» сверху или снизу. Вы тратите много времени на прокрутку большого файла. В то время как код, разделенный на маленькие файлы, легче поддается обработке, большой файл содержит код с большим количеством задач, что усложняет его понимание.

Решение

Распределите код в несколько файлов. Ваш помощник в этом — среда разработки (см. главу 20). Хорошее правило: одна область задач на один файл. Так, например, все функции по расчетам будут в одном файле, вместо того чтобы храниться возле той функции, которая им нужна. Важно логично и понятно распределять код по разным файлам.

Во многих объектно-ориентированных языках действует правило: каждый класс находится в отдельном файле. Часто даже в двух: один предназначен для видимого интерфейса, другой — для личных деталей, имплементации, не касающейся других классов. В каждом языке существуют свои традиции точного названия данных файлов.

Например, в C++ один класс закреплен в заголовочном файле (вы объявляете миру, что он существует), а в исходном файле стоит сам код. В C++ очень много традиций и мало жестких правил. Традиции в присвоении имен: они должны быть написаны в верблюжьем регистре (CamelCase), названия заголовочного и исходного файлов должны различаться только окончанием, то есть `SpaceShip.h` — заголовочный файл и `SpaceShip.cpp` — исходный.

К сожалению, в C++ существуют разные традиции. Поэтому заголовочный файл может называться также `SpaceShip.hpp` или `SpaceShip.hh`, а исходный — `SpaceShip.cc` или `SpaceShip.c++`.

Очень длинные функции

Все, что больше не помещается на экранную страницу, слишком длинное. Роберт С. Мартин в своей книге «Чистый код» рекомендует не превышать максимум из десяти строк. Все, что длиннее 50 строк, определенно слишком длинное. Тот, кто пишет функции длиннее 100 строк, в следующей жизни будет ростом 2 м 20 см и всю жизнь будет спать на гостиничных кроватях.

Проблема

Длинные функции непрозрачны, сложны для понимания и указывают на то, что автор недостаточно хорошо их продумал. Вы часто делаете больше одной задачи.

Решение

Одна функция должна решать одну задачу. Вы легко можете это проверить, объяснив задачи данной функции другому программисту. Если в этом объяснении встречается союз «и», вам стоит подумать над тем, как можно разделить эту функцию. Если у вас есть длинные функции, которые делают кучу вещей, следует записать, какие задачи за ними закреплены, и перенести относящиеся к ним части функции в отдельные функции (см. главу 15).

Подробные комментарии к блоку кода часто свидетельствуют о том, что блок, к которому относятся комментарии, лучше было бы представить отдельной функцией. Имеет смысл переносить даже отдельные строки, если они требуют объяснения и снабжены длинными комментариями.

Условия и циклы также можно вынести в отдельные функции. Тогда в случае с условиями ответвления будут стоять друг возле друга, а `if` и `else` сразу станут видны. Таким образом, функция с условиями станет прозрачной и будет лучше отражать общие связи.

Длинные расчеты для локальных переменных и отрезков кода, связывающие строчные переменные, также часто являются кандидатами для переноса в отдельные функции.

Слишком широкие функции

Как читателю этой книги, вам следует попытаться ограничиться максимум тремя уровнями отступов, а еще лучше — двумя. Однако один-единственный уровень отступа, который используется во всех строках, тоже не является добрым знаком.

Проблема

Так же, как и слишком длинные, слишком широкие функции непрозрачны, сложны для понимания и указывают на то, что автор недостаточно хорошо их продумал. Ширина не только проблема форматирования, но и показатель того, что внутренний (то есть с далеким отступом) код используется либо в слишком большом количестве циклов, либо в слишком большом количестве условий.

Решение

Часто помогают размышления над тем, как проблему можно расчленить на маленькие проблемы, которые вы, в свою очередь, перенесете в отдельные функции. В какой-то момент проблемы станут неделимыми, а их решение — зачастую простой и легкой для понимания функцией. Если код становится квадратным, то есть недлинным и нешироким, это добрый знак.

> Недобрым знаком в коде студента является то, что он становится слишком длинным. Очень просты короткие строки с небольшим количеством зависимостей в строке, но он просто запихивает все в одну. Многие переменные временные и никогда не будут использоваться снова, тогда их можно легко удалить и код станет более прозрачным. Когда студенты выучили это правило, они начали писать код размером 200 знаков на строку, в которой проводились все операции. Тут нужно найти баланс. В конце концов код приобретает квадратную форму.
>
> *Роланд Краузе, биоинформатик*

Например, следующая строка, начинающая цикл, относительно длинная и сложная для понимания:

```
for (var index = customerRecords.GetInitialValue(); index <= customerRecords.
GetLastValue(); index += customerRecords.GetIncrement()) {
    // Код, который входит в цикл
}
```

Код можно сделать квадратным двумя способами. Во-первых, вы можете добавить разрывы строк. Это работает не во всех языках, но делает код оптически менее широким:

```
for (var index = customerRecords.GetInitialValue();
    index <= customerRecords.GetLastValue();
    index += customerRecords.GetIncrement()) {
        // Код, который входит в цикл
}
```

Еще более удачным было бы разбить цикл временными переменными:

```
var start = customerRecords.GetInitialValue();
var end = customerRecords.GetLastValue();
var increment = customerRecords.GetIncrement();
for (var index = start ; index <= end; index += increment ) {
    // Код, который входит в цикл
```

Сейчас не только код стал квадратным, но и цикл — более легким для восприятия благодаря созданию временных переменных.

Многоуровневые функции if/then

Данная проблема — родственник недоброй памяти многоуровневых скобок, особенно явно она проявляется тогда, когда ход выполнения программы зависит от

двух или более условий. Сказать себе: «Итак, если A и B верны, тогда должно произойти это, если только A, то вот это, и если ни то ни другое, просто продолжить» — легко, записать это в код — тоже. К сожалению, при этом вы создаете код, который гораздо легче написать, чем прочитать.

Проблема

Если ход выполнения программы зависит от многих условий, нужно проверить их в том же блоке кода. Это очень быстро приводит к появлению многоуровневых функций, смысл которых позднее очень тяжело понять.

Пример:

```
if (test_database() == OK ){
  if (load_file() == OK ){
    /* Код для нормального состояния*/
  } else {
    /* Код для исправления ошибок load_file() */
  }
} else {
  /* Код для исправления ошибок test_database() */
}
```

Собственно говоря, это относительно простой код с тремя ответвлениями для случая, когда `test_database()` и `function_B()` выдают статус `OK`, и для случаев, когда одна из функций выдает статус ошибки. Код проблематичен по многим причинам.

- Ответвления для «`test_database()` выдает статус `OK`» и «`test_database()` выдает статус ошибки» разделены вставкой.
- Значение, выдаваемое `test_database()`, сначала тестируется, но статус ошибки для `load_file()` сначала исправляется.
- Изгородь из `if`/`else` нельзя охватить взглядом.

Решение

Все перестановки условий вынести в отдельные ответвления или сразу же в отдельную функцию. Хотя так код станет длиннее, он будет более прозрачным.

Так пример будет более понятным:

```
if (test_database() == OK) { parse_file(); } else {
/* Код для исправления ошибок test_database() */ }
function parse_file() {
    if (load_file() == OK ){
    /* Код для нормального состояния*/
    } else {
        /* Код для исправления ошибок load_file() */
    }
}
```

Алгоритмы тестов и исправления ошибок для `test_database()` и `load_file()` мы откорректировали путем переноса всех действий с файлом в отдельную функцию.

Всплывающие посреди кода числа

Не все числа так хорошо вписываются в десятичную систему, как, например, `var lengthMM = lengthM * 1000`. Здесь, даже не зная проблемы, можно предположить, что длина в метрах была переведена в миллиметры. Также многие бы распознали в умножении на 3,14159 умножение на π. Если же где-нибудь в коде стоит 2,71828, уже гораздо меньше читателей поймет, что речь идет о логарифмах. И что может значить число 86 400, использующееся в качестве константы при умножении? В англоязычной литературе такие числа называют *магическими*.

Проблема

Уже по прошествии короткого промежутка времени вы больше не помните, что число 86 400 означает количество секунд в сутках[1]. Если при создании пользовательского интерфейса вы задаете поле шириной 30 пикселов, а позднее понимаете, что лучше было бы, чтобы оно имело ширину 50 пикселов, вам придется найти и изменить все места, где стоит цифра 30. Если код полон магических чисел, очень легко может получиться так, что это число встречается во многих функциях. И если вы в данный момент попытаетесь изменить ширину поля и при этом не проявите должного внимания, в дополнение к этому еще и установите календарный формат месяца 50 дней.

Решение

Многие инструменты анализа кода распознают ошибку и сообщают о ней, видя появление магических чисел. Поместите ваши значения в переменные, а еще лучше, если ваш язык программирования располагает такой возможностью, — в константы, например `const secondsPerDay = 86400;`. Эти константы вы должны поместить туда, где сможете легко их найти, — это особенно полезно, если вы захотите позже их изменить. Если вы работаете в объектно-ориентированном языке, следует подумать, не будет ли лучше представить эти значения в виде свойств объекта. В противном случае у вас должно войти в привычку записывать неизменные значения в начале файла. Лучше всего не представлять редкие значения, например количество секунд в сутках, в качестве результатов чего-либо в вашем коде, а передать эту задачу компилятору или интерпретатору, если эти данные можно понять, увидев их расчет. Потом вам будет легче понимать, о чем идет речь, если в коде будет стоять `24*60*60`, а не `86400`.

Сложные арифметические выражения в коде

Так же, как и числа, появляющиеся в коде без привязки к переменным, длинные расчеты без комментариев или разбивки гораздо легче написать, чем потом понять. Особенно серьезные затруднения у читающего ваш код вызовет их проверка на предмет правильности.

---

[1] Приблизительно, однако обратите внимание на главу 17 и не создавайте календарных функций, в которых используется длина суток в виде 86 400 секунд.

Проблема

Окружающим нас людям, не отличающимся любовью к математике, читать и понимать формулы еще сложнее, чем программный код, это же относится и к формулам, записанным в виде кода. Формулы — это способ очень сильно сжатого представления расчетов. Если они становятся слишком длинными, многие начинают пропускать части кода или скрывают их. Именно при поиске ошибок это ведет к большим затратам времени.

Решение

Упростите код и/или переведите его в функцию, название которой в идеальном варианте сразу скажет вам, что в ней находится. Сложность можно уменьшить, присваивая промежуточные результаты временным переменным. Если этим переменным присвоены говорящие названия, сразу же запишите шаги расчета.

Как не нужно делать:

```
kiloWattPerWeek = wattPerYear / (52 * 1000)
```

Как нужно делать:

```
weeksPerYear = 52
wattPerWeek = wattPerYear / weeksPerYear
kiloWattPerWeek = wattPerWeek / 1000
```

Глобальные переменные

В коде появляется много глобальных переменных, то есть переменных, которые доступны всей программе, не только отдельным функциям. (Больше об области доступности переменных говорится в главе 26.)

Проблема

Многим неопытным программистам нравятся глобальные объекты или переменные, и они охотно и в большом количестве используют их для обеспечения взаимодействия между разными частями программы. Например, большинство программ требуют общих для всей программы настроек, управляемых централизованно, например данных доступа к базам данных. Неважно, должен ли пользователь иметь возможность изменять эти настройки, или они должны быть закреплены в коде, в любом случае следует убедиться, что различные части программы всегда используют одни и те же значения.

Наиболее простым решением являются глобальные переменные. Вначале они очень удобны, но увеличивают вероятность появления в коде ошибок и делают его менее удобным в работе. Конкретный недостаток: очень легко где-нибудь в функции создать временную переменную с таким же названием, как и у уже существующей глобальной, что приведет к непреднамеренным и с трудом поддающимся отладке изменениям. Несколько более абстрактный недостаток: существование глобальных переменных соблазняет программиста положиться на их надежную работу как в очень глубоких слоях программной архитектуры (например, во вспомогательных

функциях, которые устанавливают соединение с базой данной), так и в менее глубоких (например, в пользовательском интерфейсе). В перспективе все блоки кода станут зависимыми от каких-либо глобальных переменных, которые заданы в совершенно других местах. Последствием станет то, что код будет сложно использовать в других проектах, а также сложно понимать из-за смешения модулей.

Решение

Лучше всего не использовать глобальные переменные вообще, даже если они кажутся очень удобными на практике. Если информация, которую нужно добывать с трудом, требуется во многих местах, задайте функцию `get()`, которая только в первый раз выполнит операцию, затратив много времени, а затем при всех вызовах будет выдавать результат первоначальных расчетов.

Даже если вы очень смутно представляете свою программу как совокупность модулей на различных слоях (например, от самого глубинного слоя помощников в работе с базами данных и проведении расчетов до пользовательского интерфейса как верхнего слоя), вы уже сделали большой шаг к тому, чтобы стать хорошим программистом.

Ремонтный код

Результат расчетов не такой, каким должен быть, например, произошла ошибка неучтенной единицы — это означает, что результат всегда меньше или больше на единицу. Вместо того чтобы решить исходную проблему в самих расчетах, вы написали код, который исправляет неправильный результат.

Проблема

Пусть латание кода часто помогает, но это недобрый знак. Оно выдает читающему ваш код истинную проблему, а именно: вы не решаетесь заново переписать схему расчетов с целью откорректировать ее так, чтобы в итоге результат был правильным. Даже если это в том или ином случае остается без последствий, он может решить, что вы очень неохотно готовы разбираться с более серьезными проблемами.

Простой пример: вы хотите создать строчную переменную из массива, элементы которого разделены запятыми:

```
var elements = [1, 5, 7, 3];
var formattedStr = "";
for (var i = 0; i < elements.length(); i++) {
    formattedStr = formattedStr + elements[i] + ", ";
}
formattedStr = formattedStr.subString (0, formattedStr.length -2);
```

Вы создаете переменную, которая выглядит так: `1, 5, 7, 3,` , и обрезаете последние два знака, чтобы достичь желаемого результата: `1, 5, 7, 3`. В данном случае все работает, но этого можно и не делать:

```
var elements = [1, 5, 7, 3];
var formattedStr = "";
for (var i = 0; i < elements.length(); i++) {
```

```
    if (i != 0) {
        formattedStr = formattedStr + ", ";
    }
    formattedStr = formattedStr + elements[i];
}
```

(Если вам и сейчас бросается в глаза, что это еще не оптимальное решение, проявите терпение — оно будет найдено в следующем разделе.)

Если вы, как во втором случае, различаете в цикле первый и все остальные элементы, можете специально поставить на одну запятую меньше. Точно так же можно было бы проверить в функции `if`, является ли `i = elements.length()`, но это сложнее читается.

Решение

Относитесь серьезно к таким ошибкам без последствий. Вы скоро очень легко будете замечать, где подправили результаты расчетов с помощью кувалды.

Замените код в таких местах более правильным. В языках, которые используют `for each`-циклы, вы часто можете их применять, таким образом, вам не понадобится индекс массива и вся проблема исчезнет.

Собственная имплементация функций

В коде есть функции, похожие на те, которые уже есть в языке или имеют похожее название.

Проблема

Будучи начинающим, вы не знаете названия всех функций, которые существуют в используемых вами языке или библиотеке, и это нормально. Однако в процессе работы выяснится, что в вашем коде есть функции, которые делают то же самое, что и те, которые немного по-другому называются или вызываются.

Так как вы, вероятно, делаете больше ошибок, чем создатель вашего языка, и, поскольку его код проверяется другими разработчиками и опробован на практике, ваша имплементация хуже, чем та, которая уже существует.

Приведенный ранее пример, в котором из массива создавалась строчная переменная, во многих языках можно упростить еще сильнее. Вместо того чтобы кусочек за кусочком приклеивать массив к переменной:

```
var elements = [1, 5, 7, 3];
var formattedStr = "";
for (var i = 0; i < elements.length(); i++) {
    if (i != 0) {
        formattedStr = formattedStr + ", ";
    }
    formattedStr = formattedStr + elements[i];
}
```

используйте функцию типа `join()`:

```
var elements = [1, 5, 7, 3];
var formattedStr = elements.join (", ");
```

Решение

Здесь вам поможет только чтение. Лучше всего — книга «В скорлупе ореха» *(In a Nutshell)* издательства O'Reilly по вашему языку или (на втором месте) чужой код. Когда бы вы ни использовали базовые функции типа сортировки, разбивки строчных переменных, фильтры, форматированные выводы или смену типа переменных, вам следует прочитать, нет ли их уже в вашем языке (также см. главу 19).

Особые случаи

Ваш код, собственно, всегда выдает правильный результат, за одним исключением. Всегда, если в нем встречается строчная переменная, состоящая точно из 71 знака, все идет не так. Поэтому вы встроили в код запрос, который по-другому подходит к данному особому случаю.

Проблема

У этой ошибки есть причина. И с довольно высокой вероятностью эта причина приводит не только к ошибке, которую вы видите, но и к парочке других. Эта видимая ошибка должна вам помочь, но вы заклеиваете ее обоями и затем забываете о ее существовании. Кроме того, вероятно, что ваш механизм устранения ошибок вообще не работает и скоро на обоях снова появится отвратительное пятно. Рано или поздно код будет переполнен особыми случаями, которые вам придется учитывать при всех изменениях и расширениях.

Решение

Найдите исходную проблему. Устраните ее.

Неунифицированное написание

Названия одних функций и переменных написаны в верблюжьем регистре, в других использовано нижнее подчеркивание, некоторые из них на английском, некоторые — на немецком. Для отступа используются то табуляторы, то пробелы, в операторах иногда стоят пробелы, а иногда — нет, например `$i=1` и `$i = 1`.

Проблема

Либо мы видим здесь конфликт между многими программистами, либо автор годами приклеивал к своему коду сделанные в спешке дополнения. Или мыслями был уже не на работе. Невнимание к мелочам — знак того, что, вероятно, и важные вопросы были проигнорированы. Существует много разных традиций, но нет извинения тем, кто следует нескольким одновременно или придумывает новые.

Решение

Найдите в наиболее часто задаваемых вопросах к вашему языку, какие традиции в нем приняты, и ориентируйтесь на них. Или прочитайте главу 4 или 5, выберите какую-либо традицию и придерживайтесь ее. Среда разработки упрощает упорядочение подобного хаоса.

Функции более чем с пятью параметрами

Пример из файла `tcpdf.org`, библиотеки для создания PDF:

```
Image ($file, $x='', $y='', $w=0, $h=0, $type='', $link='', $align='',
$resize=false,
$dpi=300, $palign='', $ismask=false, $imgmask=false, $border=0,
$fitbox=false,
$hidden=false, $fitonpage=false, $alt=false, $altimgs=array()).
```

Проблема

Длинные списки параметров — признак того, что кто-то забыл разбить данные на объекты или массивы. Или это показатель того, что в коде недостаточно функций, которые могут запрашивать значения. Дополнительные параметры означают дополнительную работу для каждого, кто пытается вызвать функцию. Роберт С. Мартин в своей книге «Чистый код» в качестве идеального количества параметров называет ноль, один параметр уже хуже, чем ни одного, два хуже, чем один, трех уже следует избегать, а для всего, что больше трех, нужно иметь веские причины. Но, так как и во всеми признанном и часто используемом коде снова и снова встречаются функции с удивительно большим количеством параметров, мы установили границу несколько выше — на пяти параметрах.

Решение

Проверьте, может ли вызванная функция сама провести расчеты. Или вынесите части кода, создающие параметры, в отдельную функцию. Или же прикрепите к функции вместо различных отдельных параметров целый объект или массив, содержащий значения одного типа.

Например, в приведенном ранее примере можно было бы улучшить функцию, если бы программист задал логичные параметры по умолчанию для всех настроек PDF. Значения `x` и `y` были бы тогда `0`, если бы тот, кто вызывает функцию, не задал другие значения. Функция вызывается вместе не только с именем файла, но и со связанным с ней массивом, в котором содержатся отклоняющиеся настройки:

```
$settings = {
    $dpi => 600,
    $fitonpage => true
};

$img = Image ($file, $settings);
```

Дублирование кода

Идентичный или очень похожий участок кода появляется в коде много раз.

Проблема

Скопированный код неудобен на практике, так как вам позже придется делать изменения во многих местах. До наступления этого момента вы об этом обычно не помните. Кроме того, похожие блоки кода запутывают читающего, как города, построенные под линейку, не способствуют ориентированию. Вы тратите много времени на поиск нужного места в коде.

Решение

Копирование участков кода — знак мыслительной лени. Конечно, в небольших количествах она полезна, и поэтому здесь действует правило трех[1]. Все в порядке, если вы использовали код два раза. Но как только вы поймали себя на том, что тот же самый код копируете в третий раз, вам следует подумать над унификацией.

Начните унификацию с переноса общего инструмента, использующегося во многих функциях, в отдельную функцию. Эта функция будет вызываться предыдущей функцией, в которой теперь стоит лишь код, созданный для решения текущей задачи.

Сомнительные названия файлов

Файлы, в которых находится программа, называются `SuperActionMasterMainLoop`, `AllMiscellaneousHelperFunctions` и `ExtraMagicHacks`.

Проблема

Названия файлов как оглавление в книге. Если оглавление не работает, читающему приходится с помощью кропотливой ручной работы восстанавливать структуру всей программы. Кроме того, названия функций и переменных в таких файлах не намного лучше.

Решение

Прочитайте главу 5 и принимайте во внимание сказанное там.

Лабиринт чтения

В большинстве современных языков нет необходимости располагать функции в определенной последовательности, поэтому велико искушение записывать их

---

[1] Впервые упомянуто в: *Фаулер М.* Рефакторинг: улучшение существующего кода. — М.: Символ-Плюс, 2008.

там, где сейчас как раз открыт текстовый редактор. Поэтому родственные функции, записанные в разное время, могут находиться в разных местах.

Проблема

При чтении кода сложно понять, где начинается программа и что за чем следует.

Решение

Измените структуру программы так, чтобы относящиеся к одной задаче части кода располагались рядом. При этом придерживайтесь зависимостей: если для одной функции нужны три другие, легче понять код, не тратя много времени на поиски этой функции. Расположите код примерно по степени его значимости. Часто в программе во вспомогательных функциях есть скучный код, который должен быть, но никого не интересует, так как он, например, только форматирует и связывает строчные переменные. Уберите этот код в самый конец файла. Тот же код, который требуется где-то еще, например интерфейс какого-либо класса или объявление функций, которые должны быть видны, следует расположить в начале файла или блока кода. Многие функции появляются парами: например, одна инициализирует структуру или объект, другая снова делает его неинициализированным. Или одна функция задает переменную, а другая возвращает ее значение в исходную точку. Эти функции должны стоять рядом, так как изменения в одной функции часто требуют изменений в другой и ее непосредственная близость напоминает об этом.

Бесполезные комментарии

Комментарии к коду типа `/* сам не знаю, почему это блок тут должен стоять, но если его убрать, не работает */` показывают, что автор сам не понял свой код и не приложил достаточных усилий, чтобы его понять. Непонятный код часто содержит ошибки, которые в ходе нормального выполнения программы еще не проявились.

Проблема

Этот особенно недобрый знак производит впечатление призыва программиста к миру подсказать ему выход из положения. Но рядом никого нет, поэтому из страха испортить код он решил его не трогать. Однако страх перед собственным кодом практически гарантирует, что этот код неправильный.

Время от времени такие комментарии дописывают другие программисты, которые читают код позже (`/* что тут происходит, и, главное, почему? */`). Это признак того, что код сложен для понимания, но не обязательно он плохой.

Решение

Если вы находите в коде такие комментарии, начните сомневаться. Если нужно писать код дальше, внимательно прочитайте его, чтобы понять код лучше, чем его

автор. Если есть возможность, вам следует переписать код, чтобы он работал, *потому что* выполняются определенные условия, а не *несмотря на*...

Очень большое количество базовых классов или интерфейсов

Проблема актуальна для основанных на базовых классах объектно-ориентированных языков типа C++ или Java. Один класс должен содержать не более двух базовых классов или интерфейсов. Классы пользовательских интерфейсов могут в случае необходимости содержать четыре, но это должно дать вам повод следить за кодом с повышенным вниманием.

Проблема

Этот недобрый знак является признаком того, что вы могли бы еще раз обдумать дизайн классов. Проблемы, которые приносит вместе с собой плохой дизайн, очень многообразны и делают изменения в коде безосновательно сложными.

Решение

Если вы убедились, что какой-то класс действительно выводится из другого класса, вам поможет то, что вы громко произнесете названия обоих классов и вставите между ними «— это». Если при этом появятся странные предложения вроде «Машина — это колесо», следует воздержаться от наследования. Вместо этого, возможно, стоит задать прошлый базовый класс как свойство объекта. Связка такого типа отношений — «есть»: «У машины есть колесо».

Очень большое количество переменных или свойств объекта

Проблема

Класс очень сложен в понимании и работе. Когда начинается «очень много» и «слишком много», вы не можете последовательно работать. Если вы работаете с кодом через определенные промежутки времени и не можете разобраться в большом количестве функций и переменных, значит, их слишком много.

Решение

Сначала вам следует проверить, нельзя ли разделить класс на несколько классов. Может быть, класс пытается решать много задач, которые могут быть лучше выполнены отдельными, специализированными классами? Проверьте, возможно, определенные свойства объекта имеют смысл, лишь когда они сгруппированы. Часто они появляются вместе в других местах. В таком случае стоит объединить их в один класс, в который часто можно переместить также часть переменных.

Блоки кода и функции, помещенные в комментарии

Проблема

Такие блоки и функции — признак того, что в одной из функций скрываются нечетко определенные ошибки либо рефакторинг не был доведен до конца (см. главу 15).

Решение

Ничего страшного, если вы сначала скопировали функцию, затем одну версию поместили в комментарии, а другую переделали. При появлении непонятных ошибок вы можете вернуться к первоначальной версии. У вас будет механизм быстрой проверки того, есть ли в новой версии ошибки или не внесены ли в базу данных, которая творит бесчинства, неправильные данные. Если измененная функция не делает того, что должна, пришло время беспощадно ее удалить. Если у вас есть система управления версиями, вы будете это делать с большей готовностью, так как всегда можете откатить программу в последнее рабочее состояние.

Использование определенного браузера

Если вы когда-нибудь работали с HTML, CSS или JavaScript — базовыми составляющими каждого веб-приложения, то знаете, что разработчики разных браузеров по-разному работают с одними и теми же вещами, частично даже неправильно закладывают стандарты. Это стало причиной повышенного давления у целого поколения программистов и вызывает неприятные затруднения при настройке веб-приложений для всех важных браузеров (хотя положение вещей явно улучшилось). Многие разработчики идут легким путем и пытаются предписать своим пользователям определенный браузер.

Проблема

Это раздражает пользователей, которым не разрешено пользоваться данным браузером (например, на рабочем месте), тех, в операционной системе которых он не работает, или тех, кто просто использует один браузер, так как полностью доволен своим выбором.

Решение

Используйте валидаторы! Они существуют для всех важных веб-технологий (см. главу 13). Не скупитесь на время, протестируйте свой код в самых важных браузерах. Не пытайтесь на основании предпочтений — своих и друзей — понять, какие браузеры нужно проверить, а используйте статистику, прикрепленную в виде ссылки к статье «Браузер» в «Википедии». Не срывайтесь в другую крайность — не пытайтесь приспособить свой код ко всем браузерам. Поддерживая редкие браузеры, вы лишь изматываете себя и тратите свои силы не в том месте.

Чтобы уравнять различия в работе JavaScript в разных браузерах, вам следует использовать библиотеки, например jQuery, Underscore или Prototype. Несмотря на то что приблизительно с 2010 года имплементация JavaScript в разных браузерах похожа, по-прежнему существуют различия, к тому же вам, возможно, придется поддерживать и старые браузеры.

Существует целый ряд библиотек для JavaScript, которые созданы лишь для того, чтобы устранять различия между браузерами и поддерживать старые версии, они носят название polyfile. Если вам нужна определенная функция в браузере, следует поискать в Интернете информацию о том, можете ли вы для пользователей более старых браузеров применять такую библиотеку.

Подозрительные звуки клавиатуры

Каждые 30 секунд вы ударяете по клавишам клавиатуры, удары ритмично повторяются.

Проблема

Это указывает на то (и не только вам, а всем присутствующим), что вы делаете вручную что-то, что лучше было бы выполнять с помощью скрипта или команды Найти/Заменить. У решений, которые производят меньше подозрительных шумов на клавиатуре, много преимуществ: вы не можете случайно внести ошибку, сделав опечатку, это благоприятно влияет на самоуважение и, в отличие от совершенно механических процедур, вероятно, вы чему-то при этом учитесь. Кроме того, этот звук означает, что человек, который является его причиной, не знает чего-то совершенно элементарного, например, как можно пометить в комментарии много строк за один раз.

Решение

Потратьте пару минут и выясните, как ту же самую проблему решают другие люди. В большинстве случаев вы выясните, что у инструмента, с которым работаете, именно для этого случая есть чрезвычайно удобная функция.

JavaScript-разработчик Дуглас Крокфорд[1] отвечает на вопрос «Вам встречался код, который поначалу выглядел сумбурно, но после чистки вы понимали, что на самом деле он хорош?» так:

> Нет, такого никогда не случалось. Мне кажется, очень сложно небрежно написать хороший код (под хорошим кодом я понимаю читаемый). На этом уровне совершенно неважно, что означает этот код для машины, если я не могу понять, что он должен делать; он может оказаться удивительно эффективным, или компактным, или потрясающим еще в каком-то смысле, но это уже неважно[2].

---

[1] Дуглас Крокфорд также является автором книги JavaScript: The Good Parts.

[2] *Сейбел П.* Кодеры за работой. — М.: Символ-Плюс, 2011. — С. 107.

Даже если вы исправляете только самые явные ошибки или в дальнейшем будете избегать их, а мелкие ошибки, скрывающиеся за ними, игнорировать, вы уже многого достигли. Поэтому стоит проверить собственный код на наличие описанных здесь коричневых M&M's. Во-первых, те, кто будет читать ваш код, перестанут замечать с первого же взгляда, что перед ними плохо продуманная поделка, во-вторых, после принятия предложенных ремонтных мер речь уже не будет идти о плохо продуманной поделке. Код перейдет как минимум на ступень качества «поделка средней степени продуманности».

15 Рефакторинг

> Рефакторинг — это когда замечаешь, что foobar — довольно идиотское имя для функции, и переименовываешь ее в sinus.
>
> *Андреас Богк, цит. по: Лутц Доннерхаке. Термины информатики*

В главе 14 речь шла о признаках сомнительного кода, которые бросаются в глаза уже при его беглом просмотре. Но даже если код и на первый, и на второй взгляд выглядит совершенно чистым, некоторые изменения все равно могут пойти ему на пользу. Как новичок, вы поначалу радуетесь, что написали программу, которая запускается и делает примерно то, что должна. Но эта радость длится недолго. Вы выясните, что применение вашей программы можно было бы распространить и на другие задачи. Начальник — если вы работаете программистом — будет ждать расширений. Ведь из прототипа должна получиться следующая версия программы, пригодная для использования (надеемся, вы будете сопротивляться такой перспективе). Или вы осознаете, что с самого начала лучше было использовать другую библиотеку.

Кроме того, всякий код подвержен старению, как бы странно это ни звучало в отношении цифровой информации. Хотя он и не станет в один прекрасный день негодным и разбитым, как насквозь проржавевший велосипед, но со временем польза от него уменьшится. Во-первых, изменятся окружающие условия, во-вторых, из-за изменений смажется изначальная задумка. То, что еще вчера было правильным и осмысленным, сегодня является таковым только отчасти. Более новые области программы уже не так хорошо взаимодействуют с более старыми, красная нить, которая, хочется надеяться, некогда пронизывала программу, может с течением времени потеряться. Последний и, пожалуй, наиболее важный для читателей этой книги фактор: пока вы не самый хороший и опытный программист, вы делаете успехи сравнительно быстро. Поэтому часто случается так, что вы смотрите на код, который еще вчера казался вам вполне добротной работой, и краска стыда заливает ваше лицо...

Все перечисленное — достаточные основания для изменения существующего кода. Такое «техобслуживание» кода называется рефакторингом, и это не то же самое, что отладка. Отладка означает лишь устранение конкретных ошибок, тогда как задача рефакторинга — уберечь вас от будущих ошибок или по меньшей мере облегчить дальнейшую разработку программы. Рефакторинг сам может привести к появлению новых багов, поэтому не следует вносить изменения в код, если из-

вестно, что он содержит баги и проблемы. Рефакторинг не должен затрагивать уже имеющиеся баги, потому что тогда больше нельзя будет различить, попал ли баг в код в результате рефакторинга или был там и до этого.

Основная цель рефакторинга не в том, чтобы приукрасить код в соответствии с личными предпочтениями. Прибегайте к рефакторингу только в том случае, если хотите исправить с его помощью конкретные вещи. После рефакторинга код должен стать по сравнению с предыдущим состоянием:

- более удобочитаемым;
- более обобщенным;
- как можно более коротким.

Новый, лучший код легче понять, потому что он более удобочитаемый. Его легче применить для новых задач, потому что он более обобщенный. И в нем меньше ошибок, потому что он короче[1].

Писать ли заново?

Во многих ситуациях программист оказывается перед выбором: или начать все с чистого листа и полностью переписать код, или исправлять его строка за строкой. Велик соблазн полностью отвергнуть написанное ранее и начать все заново — с ходу отбросить жалкие попытки вылечить отдельные симптомы, а вместо этого сразу заняться сотворением нового прекрасного мира, создать код, своим великолепием подобный кристально чистому алмазу! Это естественный порыв, которому подвержены как бойцы-одиночки, так и целые фирмы. Увы, часто он приводит к краху. В старом коде трудно разобраться не только потому, что раньше его автор программировал еще хуже, чем сейчас, но и потому, что он вынужден был учитывать множество особых случаев и дополнительных условий и подстраиваться под имеющуюся программную среду. Гиена, голый землекоп и паук-птицеед не пользуются репутацией животных-красавчиков, но на то есть свои причины. Все они приспособились к определенным требованиям и условиям окружающей среды. Старые ограничения будут влиять и на новый проект и приведут к тому, что некоторые прежние непотребства сохранятся и в новой версии.

Мнения профессиональных программистов и авторов умных статей по поводу того, стоит ли начинать все сначала, разделились. В то время как одни просто удаляют плохой код и пишут лучший, другие от этого предостерегают. Джоэл Спольски, один из известнейших блогеров, пишущих о разработке программного обеспечения, считает так: «То, что новый код будет лучше старого, — абсурдное представление. Старый код использовали. Его тестировали. В нем нашли и устранили огромную кучу багов»[2]. Согласно Спольски, вообще нет оснований для предположения, что со

---

[1] Более короткий код содержит меньше ошибок, чем длинный, и не только потому, что просто занимает меньше места: см. blog.vivekhaldar.com/post/10669678292/size-is-the-best-predictor-of-code-quality.

[2] *Спольски Д.* Грабли, на которые не стоит наступать. Ч. I (*Spolsky J.* Things You Should Never Do, P. I) // www.joelonsoftware.com/articles/fog00000 00069.html.

второй попытки код выйдет лучше, чем с первой. Правда, он говорит о коде по меньшей мере среднего качества, написанном профессиональными программистами, а не о тех неприглядных дебрях, которым посвящена эта книга.

Новички в деле программирования действительно могут рассчитывать на то, что со второго раза код у них выйдет лучше, но все же, надеясь получить новый, гораздо лучший код, они во многом принимают желаемое за действительное. Поэтому мы советуем браться за написание совершенно нового кода только в том случае, если старый содержит ошибки и вы даже после тщательных поисков не можете выяснить, где они затаились. Во всех остальных случаях лучше потратить время на постепенные изменения, потому что таким образом вы тоже придете к цели, только риск, что вообще ничего не получится, будет меньше.

Когда нужен рефакторинг

В отличие от отладки, для рефакторинга не существует конкретных поводов, делающих его необходимым. Если вы отложите рефакторинг, программа не перестанет работать и вы по-прежнему сможете добавлять в нее новые возможности. Поэтому довольно сложно разработать простые правила относительно того, в каком случае стоит вносить изменения в существующий код.

В основном справочники по разработке программного обеспечения дают следующую рекомендацию: рефакторинг нужен всякий раз, когда натыкаешься на плохой код. Для новичков это может стать проблемой, так как они постоянно наталкиваются на что-то, что нуждается в исправлениях. Хотя рефакторинг и дает приятное ощущение, что вы все расчистили и прибрали, необходимо соблюдать разумное равновесие между написанием нового кода и наведением лоска на уже имеющийся. Лучше всего заниматься рефакторингом, когда прошло сравнительно немного времени с момента написания кода, но и не совсем мало: вы еще в какой-то степени помните важные детали, но уже не держитесь руками и ногами за плохие решения только потому, что на их придумывание было затрачено много сил.

Для больных с синдромом Плюшкина существует рекомендация, касающаяся посещения комнат собственной квартиры и состоящая из двух частей.

1. Не ухудшай ничего еще сильнее.
2. Каждый раз, когда ты заходишь в то или иное помещение, произведи какое-нибудь маленькое улучшение.

Эти правила применимы и для рефакторинга. Многие части кода вы никогда не встретите, если просто не будете заходить в соответствующее помещение. Это нормально, так как если у вас ни разу не возникло причин спускаться в эти глубокие подземелья, значит, хранящийся там код, возможно, и ужасен на вид, но свою задачу выполняет довольно хорошо. Следовательно, вы можете потратить свое время на более важные вещи.

Если вам встречается код, который вы хотите расширить и приспособить для выполнения новых задач, сделайте это таким способом, который по крайней мере не ухудшит его качество. После этого код должен остаться как минимум так же хорошо структурированным и таким же читаемым, как раньше.

Поэтому не нужно отдельно встраивать в код различные особые случаи, отвечающие новым нуждам. Вместо этого сделайте решение более обобщенным везде, где это возможно. Если вы, просматривая блок кода, наталкиваетесь на разделы, которые вызывают мысль: «Вот это можно было бы написать яснее», и если вы вдруг еще и знаете, какими средствами внести эту ясность, то не медлите, а сразу переписывайте. Вы в будущем поблагодарите себя за это. Соответствующим образом измените и комментарии к коду, если их не было — напишите.

Подстраивайтесь под правила присвоения имен и форматирование, используемые в данном разделе. Если они противоречат вашему нынешнему стилю, измените весь раздел. Между этими двумя возможностями лежит путь выборочного наложения заплат: вы быстро приделываете пару новых строчек в стиле, который предпочитаете сейчас, а оставшуюся часть функции оставляете неизменной. Избегайте этого третьего пути.

По возможности укажите дату изменения функции, это позволит вам проще отличить старый код от кода после рефакторинга, если вы снова будете просматривать этот файл. Если вы дисциплинированно пользуетесь системой контроля версий, то от записывания дат внесения изменений можно отказаться, а вместо этого заглядывать в ее историю. Но это работает только в том случае, если вы часто регистрируете внесенные изменения в системе контроля версий, если же делаете это только раз в месяц, отследить их будет трудно.

Новое решение не обязательно сразу же покажется более привлекательным, чем старое. Пускай это вас не пугает. Если вы и ваш код движетесь в верном направлении, но результат перестроечных работ пока что вас не удовлетворяет, это значит лишь то, что код переживает необходимую промежуточную фазу — нечто вроде стадии личинки.

Признаки того, что в данный момент внесение изменений не будет ошибкой

- **Труднообъяснимый код.** Можете задать себе следующий вопрос: «А смог бы я объяснить суть этой функции другому программисту за минуту? А за пять минут? Или хотя бы за десять? Или вообще не смог бы?» Если наиболее вероятными кажутся два последних варианта, то функцию стоит упростить.
- **Плохие признаки.** Код содержит очевидные проблемы, описанные в главе 14.
- **Вещи, вызывающие раздражение, накапливаются.** Некоторые авторы агитируют за использование правила, родственного описанному в главе 14 правилу трех раз: если вы уже в третий раз испытываете раздражение, встречая какое-либо недостаточно оптимальное решение, велика вероятность того, что вы еще много раз будете раздражаться из-за него. Это хороший повод устранить причину раздражения.
- **Неправильные места.** Вы ищете файл или часть кода в определенном месте, но ничего там не находите. Наверняка это самое естественное место для того, что вы ищете, и лучше всего было бы это туда и переместить. С одной стороны, тогда вы быстрее найдете это в следующий раз, а с другой — ячейки, в которых мозг ищет что-либо в первую очередь, зачастую и в самом деле являются для этого наиболее логичными и подходящими.
- **Проблемы с пониманием.** Перечитывая собственный код, вы сначала его не понимаете, но после некоторого размышления понимание все же наступает.

Используйте это вновь возникшее понимание и перепишите код так, чтобы при следующем прочтении у вас был шанс понять его сразу. Это может означать, что вам следует предпочесть менее элегантное, но более понятное решение. Возможно, когда вы в первый раз писали этот участок кода, то уже стояли перед таким же выбором и с гордостью избрали элегантный сложный вариант. Но мудреный код способны написать многие. Самая сложная задача — написать такой код, который вы сами все еще можете понять при повторном прочтении. Лучше тешить свое тщеславие этим.

- **Слишком много специальных решений.** Начиная встраивать в программу новые возможности, вы уже знаете, что в будущем вам придется вносить дальнейшие изменения снова и снова во многих местах. Возможно, вы даже достаточно предусмотрительны для того, чтобы написать комментарий, в котором напомните об этом будущему себе. В этом случае вам следовало бы объединить все подобные места в одну функцию (см. далее раздел «Объединяйте код»). Если в прошлом вы пренебрегли последним пунктом и сейчас вынуждены делать похожие изменения и исправления в разных местах своей программы, то воспользуйтесь этим поводом и сведите такие места воедино.
- **Ошибки возникли из-за того, что вы забыли что-то, о чем хотели «просто никогда не забывать».** Устраните повод для завязывания таких узелков на память. Люди не думают о том, о чем обязательно следует подумать, даже тогда, когда на карту поставлена их жизнь: распространенной причиной несчастных случаев даже с самыми опытными скалолазами является плохо завязанный страховочный узел. Поэтому скалолазам рекомендуется проверять своих напарников. Но и об этом вновь и вновь забывают. Не для всех, но для очень многих проблем в жизни существует лучшее решение, чем «об этом я просто должен обязательно думать каждый раз».
- **Плохое соседство.** Чаще всего к плохому коду обращается тоже именно плохой код, просто потому что обычно они написаны одним автором в одно и то же время. Если вы обнаружили подозрительное место и произвели рефакторинг, посмотрите, какие функции к этому месту обращаются, и бросьте на них критический взгляд — возможно, там прячутся области, которые также следует подвергнуть рефакторингу.

Теперь, прежде чем вы кивнете и приметесь за дело, мы просим вас еще ненадолго углубиться в себя и подумать. Понимаете ли вы, на что влияет код, который вы собираетесь подвергнуть рефакторингу? Если нет, то это может иметь неприятные последствия (см. врезку «Код это или мусор?» в конце этой главы). А есть ли у вас время на то, чтобы не только разобрать работающую систему на части, но и снова привести ее в рабочее состояние? Жизнь после незавершенного рефакторинга может стать еще тяжелее и отвратительнее, чем до него, точно так же, как после ремонта в квартире, законченного наполовину. Оба утверждения звучат тривиально, но их не так-то просто придерживаться. Если вы поняли, что совершенно не хотите переписывать код, если вы постоянно прерываетесь из-за встреч и других проектов или вам не хватает времени, то вам придется расставить приоритеты и решить, заниматься ли сейчас рефакторингом кода или разработкой новых возможностей.

В вопросе о том, стоит ли проводить рефакторинг в условиях нехватки времени, мнения расходятся.

> Это абсолютно неподходящий момент, так как вероятность сделать ошибку возрастает и вы можете потеряться в хаосе наполовину прошедшего рефакторинг кода, в котором появились новые занятные ошибки, в то время как на самом деле вам нужно выполнить кучу другой работы.
>
> *Йоханнес Яндер*

> Я принципиально не согласен со всеми аргументами, которые сводятся к тому, что не нужно делать правильные вещи, когда царит цейтнот. Именно тогда их и нужно делать! Потому что передовые методы для того и существуют, чтобы экономить время. Конечно, для этого нужны крепкие нервы и умение концентрироваться — это проблема. Но если у вас нет ни того ни другого, то лучше вообще не прикасаться к коду.
>
> *Ян Бёльше*

Компромиссная позиция, которой можно придерживаться в реальной жизни, состоит, вероятно, в том, чтобы вкладывать в рефакторинг немного больше времени, чем хотелось бы в условиях его нехватки, но все же меньше, чем нужно было бы для полного счастья.

Все по порядку

Если вы целенаправленно решили заняться рефакторингом в принципе работающей части программы, то имеет смысл сначала составить что-то вроде описи. Выпишите проблемы вместе с именами соответствующих функций. Также запишите, что вы хотите улучшить и почему. Вы должны иметь ясное представление о том, что и по каким причинам хотите улучшать, — чешите только там, где чешется. Если вы начнете рефакторинг программы одновременно со всех концов, то скоро утратите сфокусированность и не сможете улучшить ничего.

Такая опись лишь обзор, она не должна быть слишком подробной, так как сам процесс записывания важнее конкретного результата. То, что вы записываете проблему, помогает вам конкретнее сформулировать ее и лучше понять, это служит одновременно и напоминанием, и обещанием самому себе. И если все наши увещевания были напрасны и вы, доведя рефакторинг до половины, уехали на шесть месяцев в Австралию, благодаря этим заметкам у вас хотя бы будет шанс снова заставить программу работать.

Неудавшийся план

По глупости статьи в блоге Riesenmaschine были размещены в таком формате, который хотя и выглядел похожим на XML, но им не являлся. Был получен отдельный текстовый документ, содержащий несколько строк, описывающих замечательный план изменений, озаглавленный следующим образом.

План преобразования файлов статей в правильный XML и дальнейшего перехода на SimpleXML [1]

Сначала переписать использовавшуюся до сих пор инструментальную программу таким образом, чтобы она переваривала и могла создавать файлы в старой и новой форме (сделано) [2].

Затем изменить все файлы, причем так, чтобы они не могли быть дважды изменены по ошибке [3]:

— `<?xml version="1.0" encoding="iso8859-1" ?>` вставить [4];

— вокруг статьи что-то вроде `<article> </article>`;

— `&amp;` вместо & [5].

Затем переход на SimpleXML.

Все обработанные данные должны пройти через `utf8_decode` из-за наличия умляутов.

[1] Этот заголовок напоминал о том, для чего, собственно, было нужно это преобразование, и указывал тем самым на мотивацию. SimpleXML — это расширение PHP, которое сулило большие упрощения в работе с файлами статей.

[2] В изначальной версии каждый элемент псевдо-XML-кода должен был стоять в своей строго определенной строке файла, чтобы быть распознанным. Вряд ли необходимо говорить о том, что это не было хорошей идеей. На всякий случай мы все же говорим: это плохая идея. Не повторяйте это дома.

[3] Тот факт, что в конце этой строчки уже не стояло слово «сделано», указывает на то, что речь шла лишь об очень сложном и запутанном мероприятии по поиску и замене с многочисленными вероятностями допустить ошибку. Программист получше был бы в курсе, что такие задачи не решаются с помощью поиска и замены, а вместо этого пишется маленькая программка, которая считывает статьи в старом формате и преобразует их в новый. Нужно приостановить машину, для пробы пропустить через конвертер пару статей, а затем обработать с его помощью весь массив данных.

[4] Не слишком хорошая идея, UTF-8 подошла бы лучше (см. главу 17, раздел «Кодирование знаков»).

[5] & является в XML символом, зарезервированным для внутреннего использования, и поэтому не должен просто так появляться в тексте.

Таким образом, эта попытка рефакторинга, с одной стороны, не удалась, с другой — так и осталась планом. Если бы его еще раз попробовали воплотить, это привело бы к появлению двух видов статей: новых, в более правильном XML, и старых, неисправленных. Без этого документа кто-нибудь, возможно, просмотрел бы только одну из новых статей и легкомысленно решил, что у всех старых тот же формат, что послужило бы отправной точкой для новых неудач.

Прежде чем вы откроете редактор кода и начнете рефакторинг, будьте добры, зарегистрируйте нынешнее положение вещей в системе контроля версий или хотя бы сделайте резервную копию. Самое неприятное, что может случиться с вами при рефакторинге, — то, что вам внезапно придется восстанавливать вчерашнее состояние, чтобы найти неожиданно проявившуюся разрушающую базу данных ошибку, о которой раньше ничего не было известно. Чтобы обнаружить эту ошибку, вы должны воспроизвести на своем компьютере состояние системы, запущен-

ной у клиента или на вашем веб-сервере. И если вы еще не завершили изменения, связанные с рефакторингом, и у вас нет системы контроля версий, вам придется отменить все внесенные изменения. Другая опасность состоит в том, что в процессе рефакторинга вы можете дать маху, наделать ошибок, а вернуться назад уже не сможете. Во всех этих случаях лучше отменить изменения, внесенные в ходе рефакторинга, и возвратиться к проверенному практикой, хотя и безобразному, состоянию. Будучи разработчиком программного обеспечения, вы находитесь — в отличие от участника восхождения на Эверест — в весьма завидном положении, так как если вы используете систему контроля версий, то для вас не существует *точки невозврата*.

Распечатайте проблемный код и положите его вместе с примечаниями возле составленной ранее описи проблем. Эта распечатка — детальное описание существующего кода, она дополнит краткий обзор. Распечатка изначального состояния кода, положенная возле клавиатуры перед рефакторингом, — очень полезная напоминалка. Кроме того, это помогает избавиться от приступов прокрастинации, которые проявляются в том, что сначала вы просто хотите вытащить старую версию из системы контроля версий, а потом только через час закрываете вкладку с сайтом Reddit. (И если вы проигнорировали все наши добрые советы по поводу резервного копирования, то, возможно, уже через несколько часов будете благодарны за то, что у вас есть хотя бы бумажная распечатка изначального кода.)

Затем проверьте, действительно ли вы поняли, почему существует этот код и что он делает. Если вы не уверены, что понимаете цель существования и способ функционирования кода, спросите кого-нибудь, если это возможно. Объясните ему, что, по вашему мнению, делает этот код и что в нем стоит исправить. Если рядом совсем никого нет, объясните код вашему резиновому утенку (см. об этом в разделе «Если больше ничего не помогает» главы 13).

Рефакторинг всегда должен проводиться в направлении сверху вниз: сперва изменения общей архитектуры, а затем тонкости, — иначе вы потеряетесь в деталях и добьетесь немногого. Итак, если вы, например, установили, что ваша система управления пользователями выполняет и такие задачи, как обработка заказов и производство платежей, которые лучше было бы выделить в отдельный модуль, то следовало бы вначале отделить эти функции от остальных, а затем уже думать о том, будут ли в новом финансовом модуле собственные окна, меню и экземпляры базы данных. Но будьте осторожны: изменение архитектуры не означает, что вы должны сравнять с землей все здание программы с помощью бульдозера. Если вы застанете себя за вынашиванием подобных планов, отойдите от компьютера, глубоко подышите и подумайте о муравьях, переносящих туда-сюда по одной сосновые иголки. В ходе рефакторинга для всех проблем можно найти такое решение, которое складывается из маленьких автономных шагов. И ваша проблема не исключение.

Рефакторинг должен по возможности затрагивать только внутреннюю структуру программного модуля. С наружной стороны, то есть в том коде, который использует подвергаемый рефакторингу модуль, должно производиться как можно меньше изменений. Есть, конечно, исключение — случай, когда интерфейс

какого-либо модуля задан неудачно или требует расширения. Примером мог бы послужить интерфейс, который передает и ожидает в качестве параметра имя пользователя, — и вы понимаете, что на самом деле лучше было бы использовать идентификатор пользователя, так как только он является однозначным. В этом случае рефакторинг охватывает также код, который использует модуль.

Хотя все это в идеальном случае лишь мелкие изменения кода, они тянут за собой не меньшее количество ошибок, чем большие. Вероятность того, что после изменения появится ошибка, если сравнить ситуации «изменена одна строка» и «изменены пять строк», резко повышается сразу до 80 % и снова падает до 40 % при изменениях в 20 строках[1]. Причина этого в первую очередь в том, что мелкие изменения кажутся рутинными и воспринимаются недостаточно серьезно.

Поэтому после каждого шага проверяйте, все ли по-прежнему работает. Без модульного тестирования (см. главу 16) вы можете пропустить при проверке какие-нибудь особые случаи. Тогда порожденные в ходе рефакторинга ошибки выйдут наружу только в течение следующих дней или недель.

Если вы не используете систему контроля версий, но все равно не хотели бы выпускать из-под контроля последствия возникающих таким образом новых проблем, то можете оставить в коде старую версию изменяемой функции, закомментировать ее содержимое и использовать эту старую функцию только для того, чтобы обращаться к новой функции. В случае неприятности путь назад, к старому решению, будет закрыт не полностью. Но есть один недостаток: с определенной вероятностью после наступления дедлайна проекта или ослабления вашего интереса к программе старая функция так и останется в коде на веки вечные и будет сбивать с толку будущих читателей кода.

Именно такой логике мы обязаны следующим примером кода на языке C, который Йоханнес написал в начале 1990-х, в середине 1990-х расширил для новой программной платформы, а в конце 1990-х заменил обращением к одной-единственной библиотечной функции. Это было довольно халтурное сравнение двух символьных строк. Он побоялся удалить старый код, поэтому сначала закомментировал его, и даже спустя 15 лет можно обнаружить этот превратившийся в окаменелость код, который сохранился замечательно, словно внутри янтаря. Можно даже заметить — при сильном увеличении — помарки в коде:

```
int compare_strings (long firstData, long secondData) {
    /* #if TARGET_API_MAC_CARBON // расширение для новой
                                 // программной платформы
        p2cstrcpy ((char*) &str1, ((MWNickPtr)firstData)->nick);
        p2cstrcpy ((char*) &str2, ((MWNickPtr)secondData)->nick);
    #else // изначальное состояние в начале 1990-х
        BlockMove (((MWNickPtr)firstData)->nick , &str1, 255);
        p2cstr (str); BlockMove (((MWNickPtr)secondData)->nick , &str2,
        255);
        p2cstr (str);
```

[1] Эти цифры мы взяли из хрестоматийной книги Стива Макконнелла «Совершенный код» (Microsoft Press 2004, немецкое издание в Microsoft Press Deutschland, 2005).

```
    #endif
        result = strcmp((char*)&str1, (char*)&str2);
    */
    result = RelString (((MWNickPtr)firstData)->nick, ((MWNickPtr)
    secondData)->nick,
        false, false); // нынешняя версия
    return result;
}
```

В принципе, вы может производить рефакторинг хоть в текстовом редакторе с помощью поиска и замены. Но мы советуем вам везде, где возможно, пользоваться поддержкой программного обеспечения. Такие среды разработки, как Eclipse, позволяют частично автоматизировать некоторые из описанных далее операций. Хотя программа и не может сказать вам, что вы должны исправить, но в том, как это сделать, она всячески готова вам помочь: вы можете одним щелчком переместить ту или иную функцию в другой файл или класс — программа сама выполнит перемещение и произведет нужные переименования (рис. 15.1).

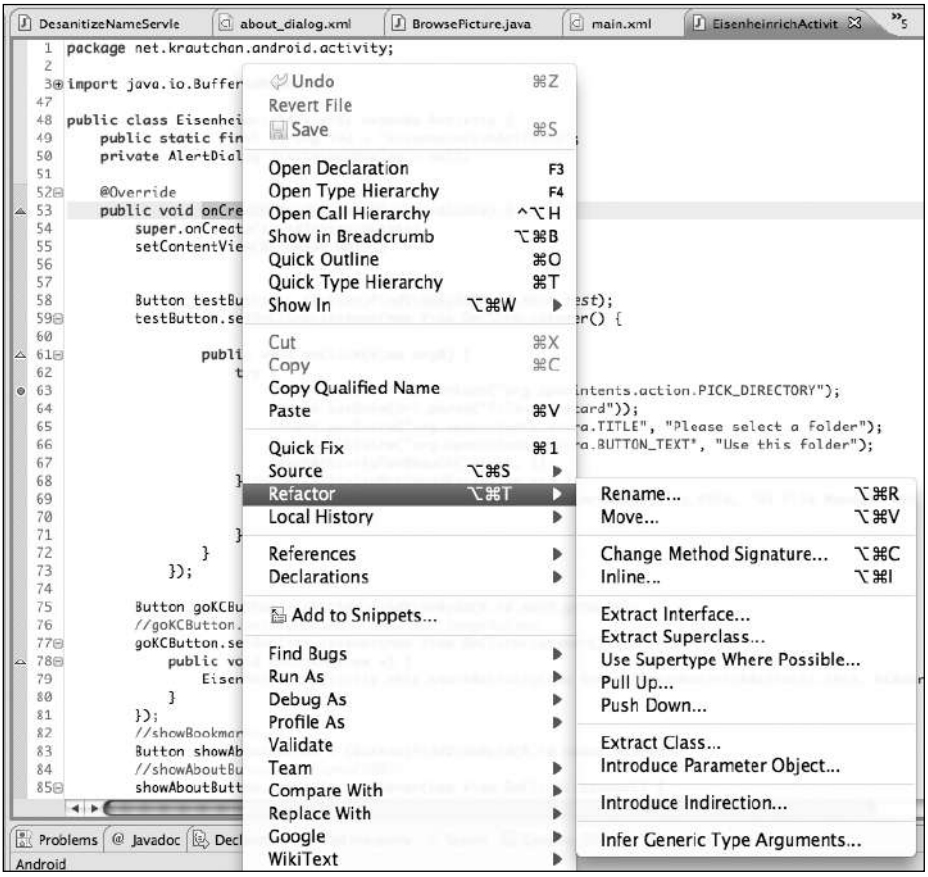


Рис. 15.1. Меню рефакторинга в среде разработки Eclipse

Подобные программные средства, помогающие при рефакторинге, существуют для различных языков и сред разработки. В «Википедии», в статье *Code Refactoring* (en.wikipedia.org/wiki/Code_refactoring), содержится их актуальный список.

Распределите код по нескольким файлам

Хорошо начать рефакторинг со смягчения ужаса перед огромной грудой кода, требующего изменений. Сгрузите устрашающие части кода, которые уже успели принести вам столько неудач, в отдельные файлы. Так вы смягчите демотивированность и страх прикасаться к собственному коду.

В Objective-C и многих других языках упаковка каждого класса в отдельный файл является общепризнанным стандартом, в Java сделать как-то по-другому трудно даже технически. Для любого другого языка это тоже хорошая идея. Попробуйте найти логические границы между частями кода. Ничего страшного, если поначалу эти части все еще будут довольно большими — на этой стадии поможет любое уменьшение.

У распределения кода по разным файлам есть и еще одно преимущество: пока весь код находится в одном-единственном файле, велико искушение реализовывать целые рабочие процессы в одной-единственной функции, будто кулинарные рецепты. Если разложить сложные рабочие потоки на отдельные файлы, каждый из которых отвечает только за одну операцию, будет легче распределить эти операции по маленьким функциям. Это следующий шаг.

Разбейте программный модуль на более мелкие модули

В теории все очень просто: неважно, идет ли речь об объектно-ориентированном, функциональном или процедурном программировании, каждый программный модуль[1] должен отвечать за одну задачу и, наоборот, одна задача не должна рассеиваться по разным модулям. Например, в научно-исследовательском аналитическом ПО экспериментальные данные должны считываться в одном модуле, анализироваться — в другом, а третий модуль должен записывать результаты в файлы.

Но при написании реальной программы все зачастую сложнее: так, для анализа вам могут понадобиться не только экспериментальные данные из вашей базы данных, но и калибровочные параметры, которые берутся из какого-либо файла. Таким образом, вам нужно также написать функции, которые будут читать эти файлы параметризации. Функции, которые записывают результаты в файлы, должны также преобразовывать эти результаты в XML, чтобы их смогла использовать другая программа, а по завершении анализа еще и должно быть отправлено сообщение пользователю. Вот так вот нечто, что вначале казалось простым, быстро может развиться в систему, где операции по считыванию информации из базы данных и операции с файлами

---

[1] Здесь под модулем мы понимаем, например, класс в объектно-ориентированном или функцию в процедурном программировании.

стоят рядом и один и тот же модуль отвечает за форматирование, вывод информации и отправку сообщения об успешном выполнении анализа. Это хотя и допустимо, но делает систему непонятной и приводит к тому, что через некоторое время вам придется постоянно искать, куда именно был помещен тот или иной элемент.

Поэтому имеет смысл из трех модулей сделать пять.

- Прежние модули:
  - модуль считывания;
  - модуль анализа;
  - модуль записи файлов.
- Новые модули:
  - модуль базы данных;
  - модуль работы с файлами (сюда перекочевывают операции считывания из первого модуля и операции записи файлов из последнего модуля);
  - модуль анализа данных;
  - модуль форматирования в XML;
  - модуль отправки сообщения.

После этого преобразования каждый модуль стал более маленьким и фокусированным. Уже сами названия модулей довольно точно дают понять, чем они занимаются. Основная программа отвечает только за «оркестровку», обращаясь к функциям из отдельных модулей в нужном порядке.

Подведем итог: разбивайте слишком *большие программные модули* или объекты на более удобные в обращении единицы, которые занимаются только одним аспектом. Для этой цели запишите, зачем нужен каждый модуль, и подумайте над тем, одна это задача или несколько. Потом создайте новые модули и распределите по ним код таким образом, чтобы похожие функции оказались вместе, а элементы, выполняющие неродственные задачи, — раздельно.

ЧТО ЗНАЧИТ «СЛИШКОМ БОЛЬШОЙ»?

Нет такого правила, которое определяет, сколько нужно строк, чтобы выразить идею. Как читатель этой книги, вы можете пока что исходить из того, что ваши функции, объекты или классы скорее слишком большие, чем слишком маленькие.

Обращайте внимание на повторение одинаковых или похожих частей кода

Когда вы пишете длинные блоки кода, в них почти всегда можно обнаружить области, повторяющиеся много раз, — хоть не обязательно все они находятся в одном и том же месте, но они разбросаны по программе. Этих повторений можно избежать, если переместить такой код в функции и обращаться только к этим функциям.

«Заголовочный код» намекает на то, что был бы не прочь отправиться в функции

Если вы обнаружили код, который разделен на отдельные шаги многочисленными заголовкоподобными комментариями, то часть вашей работы уже сделана. Вероятно, эти заголовки даже подсказывают вам, как могли бы называться новые функции.

Активно ищите подпроблемы, которые не имеют отношения к непосредственной задаче кода в этом месте

Функция, которая считывает данные из базы данных, тут же конвертирует эти данные в HTML? Тогда в большинстве случаев имеет смысл вынести преобразование в HTML в отдельную функцию. Подробнее об этом поговорим в следующем разделе под названием «Устраняйте побочные эффекты».

Разделяйте код на более короткие функции

Функции служат для структуризации кода, а структуризация важна по крайней мере не меньше, чем возможность повторного применения, просто потому, что она значительно облегчает чтение. Вводите функции даже там, где происходит что-то совершенно особенное, больше нигде не применяемое. К тому же потом часто выясняется, что те вещи, повторное применение которых казалось при их написании стопроцентно невозможным, все же могут пригодиться в других местах.

Среда разработки (см. главу 20) может облегчить эту задачу. Во многих средах разработки есть функция Выделение метода, которая автоматически отправляет выбранную область кода вместе со всеми требующимися параметрами в отдельный метод и иногда даже делает предложения по поводу того, какие другие области кода также можно было бы заменить с помощью этого метода.

Такое обособление отдельных задач в коротких, автономных единицах — это важный шаг, который является основой для дальнейшего рефакторинга, так как вы только в том случае сможете разумно пересобрать свой код заново, если он не хранится в бесконечно длинных функциях. Кроме того, дублирование кода легче обнаружить, если этот код не спрятан где-то на глубоком уровне индентации.

Но все же имейте в виду, что один только размер модуля еще ничего не говорит о том, достаточно ли этот модуль фокусированный. Простые задачи решаются обычно с помощью небольшого количества кода, сложные, напротив, требуют большого количества отдельных команд. Программы с графическим интерфейсом пользователя особенно часто имеют довольно большие модули для выполнения соответствующих задач. Хотя код для внутренней обработки данных может быть гораздо более сложным, в конечном счете он занимает намного меньше строк, чем код графического front-end.

Поэтому не бойтесь писать модули, заметно превышающие по размеру остальные. Хорошо продуманный модуль воплощает какую-либо совершенно определенную идею, не больше, но и не меньше.

Устраняйте побочные эффекты

Нужно приложить некоторые усилия, если вы хотите структурировать код таким образом, чтобы он состоял из коротких функций, каждая из которых занимается только одним делом. Чаще всего именно новички грешат тем, что просто записывают код в соответствии с запланированным ходом выполнения программы, лишь мимоходом вставляя там и тут функции и вынося в них те или иные части кода.

Если функция `analyzeExperimentData()` анализирует экспериментальные данные и тут же еще и записывает их в файл, то сохранение — побочный эффект. По-

бочные эффекты — это проблема, так как они затрудняют понимание кода и усложняют его сопровождение. Иногда такие функции с побочным эффектом можно распознать по комментариям вроде: «Внимание, КРОМЕ ТОГО, эта функция делает еще и...» К функции без побочных эффектов можно обращаться сколь угодно часто, и обращение к функции можно при необходимости переместить в другое место. В особенности те функции, которые возвращают значение, должны делать это без побочных эффектов.

В нашем примере из имени функции понятно, что на самом деле должно быть ее задачей. Может показаться, что с помощью этой функции удобно выполнять также и сохранение, но тогда ее придется тут же переделывать, если вдруг окажется, что результаты лучше было бы сохранять в базу данных. Или вы скопируете ее во вторую функцию и замените ту часть, которая записывает результаты в файл, на часть, сохраняющую их в базу данных. После этого у вас появятся две очень похожие функции, опасность перепутать которые велика.

Делайте короче!

Разделяйте код на короткие функции, которые действительно выполняют только необходимые вычисления и возвращают результат. Если вы хотите написать функцию, которая анализирует данные, то она должна возвращать результат тому коду, который к ней обратился. Затем этот код обращается к функции, которая сохраняет этот результат в файл (или в базу данных). Недостаточно просто вынести задачу сохранения в какую-либо функцию, к которой вы обращаетесь напрямую из функции, выполняющей анализ. Это лишь смещает проблему, и побочный эффект все равно присутствует, хоть и косвенно.

Один модуль — одна задача

В общем случае каждый программный модуль — неважно, класс это или функция, — должен иметь только одну задачу. Примером неудачи, ставшей результатом того, что идентификаторы фрагментов текста были перегружены дополнительными функциями, является случай, произошедший при написании этой книги. В довольно новом на тот момент редакторе *Stypi* — онлайн-инструменте для одновременной совместной работы с текстами — фрагменты текста, добавленные разными авторами, помечаются разными цветами, по крайней мере в теории. На практике текст оставался одноцветным, из-за чего было легко запутаться. Почесав затылки и переписывавшись с разработчиками, мы выяснили следующее: каждому пользователю в этом редакторе присваивается только один из 16 цветов. Этот цвет получается из идентификатора пользователя путем его деления по модулю 16, и поэтому его нельзя изменить. Единственное решение состояло в том, чтобы до тех пор создавать новые аккаунты, пока наконец всем участникам проекта не присвоили разные цвета.

Объединяйте код

Довольно часто в ходе работы над проектом вы снова и снова пишете похожие функции и используете похожие структуры данных. Поначалу это даже оправданно:

если писать код под решение конкретной проблемы, дело идет быстрее. Более обобщенное решение требует больших мыслительных усилий и временных затрат. Также во время разработки программы зачастую совсем не ясно, какую форму примет код в окончательном варианте.

Но когда программа уже, в принципе, работает, наступает время для замены похожих функций более обобщенными. Это дает несколько преимуществ. Код в целом становится короче, хотя сама написанная обобщенно функция и получается немного длиннее. Такой код меньше подвержен риску возникновения ошибок, так как в нем больше не содержится разных реализаций похожих задач. И его гораздо легче сопровождать: если вы хотите расширить функциональность, вам необходимо произвести изменения только в одном месте.

Проверьте, не встречаются ли где-нибудь похожие друг на друга функции — они не обязательно будут находиться в одном и том же классе или программном модуле. Подумайте, каким образом можно объединить их в одну более обобщенную функцию. Если это кажется невыполнимой задачей, то хотя бы примите во внимание тот факт, что в будущем вы наверняка забудете о наличии многократно встречающихся похожих функций, и напишите комментарий: «Внимание, изменить также здесь и здесь!»

Приведем пример: приложение должно загружать с сервера личные данные и данные об изображениях. Программист написал в ходе разработки приложения следующие функции:

```
function getFigures (startFigureId, endFigureId) {
    if (startFigureId >= endFigureId) {
        return null;
    }
    var numFigures = endFigureId – startFigureId;
    var figures = getViaHttp('http://www.example.com/figures/?start=' +
    startFigureId + '&amp;end=' + endFigureId);
    if (figures != undefined) {
        return figures;
    }
    return null;
}

function getPersons (startPersonId, numPersons) {
    if ((numPersons == null) || (numPersons <= 0)) {
        return null;
    }
    var persons = getViaHttp('http://www.example.com/persons/?start=' +
    startPersoIdn + '&amp;count=' + numPersons);
    if (persons != undefined) {
        return persons;
    }
    return null;
}
```

В первом случае программист решил, что функция будет ожидать начальный и конечный индексы. В этом случае она проверяет, имеют ли смысл начальный

и конечный индексы, то есть является ли конечный индекс большим, чем начальный. Если это так, то функция загружает данные об изображениях и проверяет успешность загрузки.

Во втором же случае программист решил, что функция будет ожидать начальный индекс и количество человек. Функция проверяет, было ли определено количество и является ли оно большим, чем 0, и загружает личные данные так же, как и первую функцию.

Рефакторинг функций приводит к следующему результату:

```
function loadFromWebservice (startIndex, endIndex, url) {
    if (startIndex >= endIndex) {
        return null;
    }

    var num = endIndex – startIndex;
    var data = getViaHttp(url + '?start=' + startIndex + '&amp;count=' +
    num);

    if (data != undefined) {
        return data;
    }
    return null;
}
```

Теперь обращение к этой функции могло бы происходить в программе следующим образом:

```
var figures = loadFromWebservice (startFigureId, endFigureId,
'http://www.example.com/figures/');
var persons = loadFromWebservice (startPersonId, endPersonId,
'http://www.example.com/persons/');
```

Так как *сигнатура метода* для получения личных данных изменилась, то код программы, который ранее обращался к функции `getPersons()`, также должен быть изменен соответствующим образом. Возникающая после рефакторинга необходимость одинаково обращаться к похожим функциям чаще всего является преимуществом, поскольку приводит к тому, что обращающиеся к ним участки кода также становятся более похожими друг на друга. В нашем случае код, который обращался к функции `getPersons()`, после рефакторинга будет иметь большее сходство с кодом, обращавшимся к функции `getFigures()`, чем раньше.

СИГНАТУРА МЕТОДА

Сигнатура метода — это что-то вроде интерфейса метода, то есть все, что может быть интересно внешнему наблюдателю: имя, с помощью которого происходит обращение к методу, количество и последовательность его параметров, а также тип возвращаемого значения.

Еще один результат рефакторинга в нашем примере заключается в том, что информация об URL, по которому получаются данные, вынесена за пределы функции.

В большинстве случаев это хорошо, потому что благодаря этому функция становится более пригодной для повторного применения. Если вы захотите использовать этот код для загрузки данных с разных серверов, то после такого рефакторинга сделать это будет легче.

Однако если этот эффект нежелателен, например, потому, что функция обращается к коду в совершенно разных областях приложения и вы не хотите разбрасывать переменные для URL-адресов по всей программе, то можете создать так называемые удобные методы. В коде они стоят как можно ближе к обобщенной функции и представляют собой специфические сценарии использования. В нашем примере они выглядят так:

```
function getFigures (startIndex, endIndex) {
    return loadFromWebservice (startIndex, endIndex,
    'http://www.example.com/figures/');
}
```

И так:

```
function getPersons (startIndex, endIndex) {
    return loadFromWebservice (startIndex, endIndex,
    'http://www.example.com/persons/');
}
```

Обобщение не всегда и не только преимущество. Если вы объедините две функции в одну, то при известных условиях создадите зависимость там, где ее раньше не было. Вместо двух самостоятельных функций, которые могут гибко развиваться дальше — если необходимо, в разных направлениях, — у вас теперь одна. Не нужно из-за этого отказываться от обобщения, но сначала немного подумайте о том, действительно ли между двумя частями вашего кода есть не только чисто поверхностное, но и более глубокое сходство.

Не нужно совмещать вещи насильно, если результат от этого будет менее наглядным и понятным, чем предыдущее состояние. Если вы уже можете предвидеть, что в результате планируемого вами объединения получится функция, которая, судя по передаваемым параметрам, будет заниматься очень разными вещами, то это признак того, что обобщение в данном случае принесет скорее вред, чем пользу. Например, для функции `drawShape(shapeType, posX, posY)` понадобился бы комментарий, который объяснял бы, что параметрами могут быть `circle`, `square` и `bunny rabbit`. В то время как смысл функций `drawCircle(posX, posY)`, `drawSquare(posX, posY)` и `drawBunnyRabbit(posX, posY)` понятен сам собой.

Это означает также то, что, проводя рефакторинг, иногда нужно отходить от слишком обобщенных решений. Если вы нашли функцию, которая принимает небольшое количество определенных параметров (часто два) и в зависимости от этих параметров выполняет очень разные задачи, то вам следует подумать о ее разбиении на отдельные функции. Параметры, которые управляют только поведением функции, — признак функции, которая замахивается на слишком многое.

В нашем примере в функции `drawShape()`, по сути, спрятаны три функции, которые делают абсолютно разные вещи, — что именно, определяется параметром

shapeType. Разумно было бы соединить то общее, что в них есть, в одну функцию, но при этом создать три функции — drawCircle, drawSquare и drawBunnyRabbit, каждая из которых обращалась бы к общей функции.

Понятнее оформляйте условия

В главе 14 речь шла среди прочего о необычайно запутанных и засунутых на глубокий уровень индентации условиях if/then. В ходе рефакторинга стоит критически взглянуть и на те условия, которые на первый взгляд выглядят вполне прилично.

Где стоит if, желателен и else

Хотя довольно часто и встречаются случаи, когда задумано только одно условие без противоположного, однако намного чаще при невыполнении условия должно происходить что-то другое. Если вы пропускаете ветвь else, то, вероятно, забываете позаботиться о том, что должно произойти в таком случае. Подумайте: может, вам все же придет в голову что-нибудь, что могло бы подойти для включения в ветвь else. Если нет, то все же лучше создайте пустой else и объясните в комментарии к нему, почему данный случай никогда не наступит. Так вы дадите знать самому себе будущему, что хотя бы подумали об этом.

Заблаговременно поставленный оператор return может сделать код более удобочитаемым

В языке C правилом хорошего тона считалось снабжать каждую функцию только одним оператором return. В конце выполнения функции снова освобождались ресурсы, выделенные ею для самой себя, а каждая дополнительная возможность возврата легко приводила к утечкам памяти. В языках со *сборкой мусора* это уже не ваша проблема. Правда, если вы используете какое-либо инструментальное средство анализа кода (см. главу 13), то оно может с определенным недовольством реагировать на чрезмерно большое количество точек выхода в функции, считая это признаком функции, которая хочет слишком многого сразу или слишком непонятная.

СБОРЩИК МУСОРА

Сборщик мусора — это часть среды выполнения какого-либо языка, которая обеспечивает удаление ненужных более объектов из оперативной памяти, освобождая тем самым место для новых объектов.

Псевдокод с единственным оператором return:

```
doSomething(someVariable, someOtherVariable) {
    if (someVariable != null) {
        // сделать что-то
    } else {
```

```
        // сделать что-то еще
    }
    return someResult;
}
```

Тот же код с двумя точками выхода:

```
doSomething(someVariable, someOtherVariable) {
    if (someVariable == null) {
        return false;
    }
    // сделать что-то
    return someResult;
}
```

Теперь основная часть кода (`// сделать что-то`) передвинута на один уровень индентации ближе к левому краю. На таком коротком примере это незаметно, но в идеале вам следовало бы вставлять множественные точки выхода только в начале и в конце функции, где потенциальные читатели кода готовы к их появлению, и по возможности не делать этого в середине, где они могут только запутать в случае, если речь идет о более длинных функциях. А еще лучше — не пишите длинных функций.

Ветви if и else должны быть приблизительно одинаково важны

Если это не так, то лучше с самого начала убрать с дороги тривиальные и особые случаи, как описано в предыдущем пункте. Это позволит уменьшить количество отступов и вложений.

Длинные и/или сложные строки в ветви if, ветви else или внутри условия можно вынести в функции

Тогда вся сложность переместится в функции — туда, где она не будет мешать. Добавление функции:

```
function isCuteHedgehog(someAnimal) {
    if (isHedgehog(someAnimal) && isCute(someAnimal)) {
        return true;
    }
}
```

позволит сэкономить место в той части кода, где находится условие, и избавит от лишних размышлений при ее чтении. Там будет написано только `if (isCuteHedgehog(someAnimal))`. Саму функцию вы можете поместить в «ящик для инструментов» — отдаленную часть вашего кода.

Перемещение частей условия в функции может иметь смысл даже в том случае, если оригинал был довольно краток: чтение новой версии потребует меньших размышлений, так как новое имя функции будет играть роль комментария. `if isEven(someNumber)` читается проще, чем `if (someNumber % 2 == 0)`.

Ищите идентичные строки, присутствующие во всех ветвях условия if/then

Эти строки вы можете вынести за закрывающую скобку условия. Тогда вероятные изменения в будущем понадобится производить только в этом месте.

Подумайте об использовании конструкции switch/case

Операторы `switch/case` действуют как ряд поставленных друг за другом операторов `if`. То, что при использовании `if/else` было бы запутанным нагромождением условий, часто сразу становится гораздо более удобочитаемым, когда применено `switch/case`. Дополнительное преимущество состоит в том, что конструкции `switch/case` предусматривают параметр по умолчанию (с именем `default` или `else`), который применяется здесь чаще, чем при использовании `if/else`. К сожалению, конструкция `switch/case` во многих популярных языках имеет тот недостаток, что каждый оператор `case` должен обязательно заканчиваться командой `break`, иначе он сольется со следующим оператором `case`. Это может быть преднамеренным, как в следующем случае:

```
switch (animal) {
    case 'rabbit':
    case 'kitten':
        // сделать что-то
        break;
    case 'velociraptor':
    case 'shark':
        // сделать что-то еще
        break;
    default:
        // не делать ничего
}
```

Но такой «проскок» частенько прокрадывается в код вопреки намерениям автора, и потом эту ошибку сложно найти и устранить. Дуглас Крокфорд в книге «JavaScript: сильные стороны» советует по этой причине вообще отказаться от применения конструкции `switch/case`. Если вас никогда не смущало отсутствие параметра по умолчанию, но вы уже не раз имели неприятности из-за «проскока», добавленного по ошибке, то, возможно, вам больше подходит оператор `if/else`. Решайте сами. Если вы вставляете «проскок» намеренно, то лучше всего снабдить это место комментарием.

Правильный цикл для правильной цели

Как программист не слишком опытный, вы, возможно, предпочитаете одну определенную форму цикла, используя которую чувствуете себя наиболее уверенно. Но различные типы циклов существуют не только потому, что программисты любят разнообразие. Каждый из типов лучше всего подходит для какой-либо совершенно определенной цели. Конечно, любой тип цикла можно подогнать под выполнение не свойственных ему задач, но это ненужное усложнение жизни.

Если вы с самого начала знаете, как часто должен выполняться цикл, выберите цикл `for`. Если не знаете — цикл `while`. Циклы `while` несколько более удобочитаемы, чем циклы `do-while`, потому что условие указывается в самом начале. Многие циклы `do-while` можно без проблем преобразовать в циклы `while`.

Если вы включаете в цикл `for` критерий прекращения цикла (обычно с помощью оператора `break`), то это признак того, что данный цикл лучше было бы сделать циклом `while`. Если вы используете в заголовке цикла `for` конструкцию, которая сильно отклоняется от `i = 0; i < someValue, i++`, это также признак того, что его лучше было бы преобразовать в цикл `while`. Если вы злоупотребляете счетчиками, чтобы досрочно выйти из цикла (например, внезапно делаете переменную-счетчик `i` равной 100), это также признак того, что этот цикл лучше было бы преобразовать в `while`.

Если вы добавляете в цикл `foreach` или `while` переменную-счетчик, значение которой возрастает после каждого прохождения цикла, это признак того, что цикл лучше сделать циклом `for`. Исключением из этого правила являются *безопасные счетчики (safety counters)*.

Все это лишь ориентировочные указания. Могут найтись достаточно веские основания для того, чтобы все же выбрать иной тип цикла. Тогда вам, возможно, стоит изложить эти основания в комментарии, чтобы не пришлось снова думать об этом, когда вы в следующий раз увидите этот код.

Оформляйте циклы понятнее

Как и в случае с условиями, на циклы стоит посмотреть критически, даже если на первый взгляд с ними все в порядке. Часто в них кроются большие возможности для оптимизации.

Каждый цикл должен выполнять только одну функцию

Тот факт, что цикл можно приспособить и для выполнения двух функций, не является причиной для нарушения этого правила. В связи с этим неопытные программисты часто лелеют смутные мысли об экономии. Но выяснить, действительно ли использование цикла с несколькими функциями значительно сократит продолжительность вычислений, можно только проверкой (см. раздел «Тестирование производительности» главы 16). Если вам неохота этим заниматься, то лучше используйте в интересах читателей вашего кода больше отдельных циклов, каждый из которых выполняет одну задачу.

Двух уровней вложенности достаточно

Стив Макконнелл, автор «Совершенного кода», разрешает своим читателям до трех уровней. Как читатель этой книги, лучше ограничьтесь, пожалуй, двумя. Есть исследования, которые говорят о том, что и у хороших программистов, начиная с третьего уровня вложенности, начинаются серьезные проблемы с пониманием.

С момента добавления второго уровня вложенности все счетчики циклов должны получать говорящие имена

`i` и `j` легче перепутать при написании и тяжелее правильно истолковать при чтении, чем, например, `lineCounter` и `wordCounter`.

Чем меньше строк с индентацией, тем лучше

Многие проблемы с индентацией можно смягчить, если, как описано ранее, в определенных случаях добавить еще одну точку выхода в начале функции, а не только единственный оператор `return` в конце.

Оператор continue сбивает многих читателей с толку, и его почти всегда можно избежать

Тот факт, что в программировании глагол `continue` (продолжить) означает «Прерви то, что ты прямо сейчас делаешь, и со следующего шага продолжай выполнять цикл», а `break` (прервать) — что-то совершенно другое, становится причиной долгих размышлений, а также частых вопросов в Сети с просьбой объяснить разницу между `break` и `continue`. В качестве подсказки можно предложить добавлять в конце слово `loop` (цикл): фразы `break loop` и `continue loop` лучше передают суть происходящего. Слишком большое количество выходов из цикла — признак того, что, возможно, было бы лучше разбить его содержимое на более мелкие фрагменты. Кроме того, оператор `continue` в циклах `while` часто в результате ошибки приводит к появлению бесконечных циклов. Лучше всего вообще не предусматривать выходы в теле цикла, то есть отказаться от `break` и `continue`. Дуглас Крокфорд в книге интервью «Кодеры за работой» дает следующее объяснение: «Я еще не встретил ни одного фрагмента кода, который не смог бы улучшить, выкинув оператор `continue`. Да, с его помощью легче создать какую-то сложную конструкцию. Но я заметил, что всегда могу улучшить эту конструкцию, найдя способ выкинуть его. Так что лично я никогда не использую оператор `continue`. Если же я вижу `continue` в своем коде, значит, что-то недодумал»[1].

Рутинные задачи цикла должны стоять вместе в начале или в конце

Не распределяйте их по всей средней части. Такие задачи обычно связаны с теми переменными, которые вы ввели непосредственно перед циклом (чаще всего это, например, увеличение значения переменной, продление символьной строки или массива).

Циклы while должны иметь критерий прекращения в качестве стоп-крана

По крайней мере, если вы получаете доступ к файлам или через Сеть к другим серверам. Так как вероятность ошибок при обращении к внешним ресурсам гораздо выше, чем при считывании с запоминающего устройства или жесткого диска, вам стоит ограничить количество попыток и лучше завершить программу с одной

---

[1] *Сейбел П.* Кодеры за работой. — М.: Символ-Плюс, 2011. — С. 107.

ошибкой, если этот порог превышен. Например: «Делай что-нибудь, пока *x* меньше *y*, но не позднее чем после 100 повторений — прекрати». Соответствующий код мог бы выглядеть следующим образом:

```
tries = 0
max_tries = 100
while (some condition) {
    do something
    tries++
    if (tries >= max_tries) {
        log_error ("connect failed")
        exit (1)
    }
}
```

Не вставляйте критерий прекращения в условие `while`. Условие `while` и условие прекращения цикла имеют совершенно разные задачи и должны быть пространственно разделены.

Посмотрите критически на переменные

В главе 14 уже шла речь о глобальных переменных и о том, что их стоит избегать. Но и другие переменные стоит иногда оформить еще более локально. Посмотрите на все переменные критическим взглядом: действительно ли необходимо распространять их присутствие на далекие области кода? Нужны ли они вообще? Если вам удастся ограничить количество и распространение переменных, это сделает код более понятным, так как его читателям не придется одновременно держать в голове так много элементов. (Больше информации об области применения переменных вы найдете в главе 26.)

Делайте из переменных-членов локальные переменные везде, где возможно

В объектно-ориентированном программировании иногда можно сделать из переменных-членов локальные переменные метода. Переменные-члены должны содержать только те значения, которые важны для состояния объекта и не могут быть получены путем простого вычисления. Промежуточные значения или значения, которые нужно просто вычислить (например, сумму брутто), не следует хранить в переменных-членах.

ПЕРЕМЕННЫЕ-ЧЛЕНЫ

Переменные-члены — это переменные, которые могут быть прочитаны и изменены всеми функциями того или иного объекта.

Управляющие флаги в циклах указывают на наличие потенциала для улучшения

Часто такие флаги носят имена вроде `done`, `found`, `success` или `keepGoing`, которые намекают на то, что здесь автор самостоятельно заново реализовал элементарные функции используемого языка программирования в форме флага. Это неэлегант-

но, и этого можно избежать, если предусмотреть дополнительные выходы из цикла (досрочный `return`) или вынести части кода в методы. Таким образом, вы не будете принуждать читателей кода просматривать цикл до конца, если там на самом деле ничего больше не происходит.

Если переменная никогда не изменяется, то имеет смысл сделать ее константой

С одной стороны, так код становится чище, с другой — вам больше не нужно каждый раз думать о ее состоянии. Во многих языках случайное изменение константы приводит к синтаксической ошибке, что еще на стадии программирования напоминает вам, что речь идет о константе.

Подумайте о перемещении переменных и установке их в непосредственной близости от места использования

Если все задействованные в функции переменные определены в начале, код, конечно, выглядит чище. Но в то же время читателям вашего кода придется думать об этих переменных еще до того, как им станет ясна цель их использования.

Рефакторинг баз данных

Даже если вы мало работаете с реляционными базами данных и еще меньше ими интересуетесь, тема нормализации должна быть вам знакома. Нормализацию схемы базы данных можно грубо сравнить с рефакторингом программного кода. Она дает похожие преимущества: устранение избыточности, повышение абстрактности и уменьшение возможностей ошибок.

Существуют различные степени нормализации базы данных, при этом обычно упоминаются шесть нормальных форм, от первой до шестой, а также нормальная форма Бойса — Кодда — в зависимости от того, насколько сильно схема приближается к этим передовым методам. Опыт говорит, что по-настоящему важны только первые три нормальные формы[1].

1. **Первая нормальная форма (1НФ):** графы базы данных не содержат значений, которые можно разумно распределить по еще более дробным разделам. Например, графа `Name`, которая до нормализации содержала значение `Laura Berndsen`, разделяется на графы `Firstname` и `Lastname`, графа `Address` — на улицу, номер дома и почтовый индекс. После этого вы сможете без проблем осуществлять поиск только по фамилиям, избавившись от необходимости каждый раз разделять в своей программе данные о человеке на имя и фамилию.

   Содержащиеся в записях характеристики, которые могут иметь самостоятельную значимость, тоже должны быть перемещены в собственную таблицу. Например, для базы личных данных это означает, что наряду с таблицей физических лиц должна существовать также таблица адресов, так как адреса могут быть полезны и для таблицы филиалов или поставщиков.

---

[1] Все нормальные формы подробно описаны на странице https://ru.wikipedia.org/wiki/Нормальная_форма.

Для каждой таблицы вам следует определить индивидуальный идентификационный номер (см. главу 26), который в мире реляционных баз данных называется первичным ключом (`Primary Key`), чтобы можно было создать отношения между определенными записями в разных таблицах.

2. **Вторая нормальная форма (2НФ):** выносите характеристики в отдельные таблицы, если запись содержит массив или список характеристик. Например, в случае с таблицей поставщиков запросто может оказаться, что у одной фирмы-поставщика будет несколько заводских адресов. Не следует решать эту проблему, создавая в своей схеме базы данных несколько похожих граф, например три графы с адресами. Это верный признак того, что эти многократно встречающиеся характеристики просятся в собственную таблицу.

   Во второй нормальной форме таблицы связаны отношениями. Эти отношения вы определяете следующим образом: в таблице с многократно встречающимися характеристиками (в нашем примере это таблица адресов) вы определяете одну графу как внешний ключ (`Foreign Key`), который связан с первичным ключом (`Primary Key`) другой таблицы (например, поставщиков). Тогда база данных будет знать, что все адреса с определенным внешним ключом принадлежат одному и тому же поставщику.

3. **Третья нормальная форма (3НФ):** в то время как цель 2НФ — избежать использования нескольких граф для одной и той же характеристики, искусство применения 3НФ в том, чтобы найти графы, которые, если рассматривать их построчно, содержат определенные, много раз встречающиеся последовательности, и вынести их в собственные таблицы. Например, если в базе сотрудников указан филиал, в котором работает сотрудник, то в третьей нормальной форме название филиала не стоит в графе таблицы сотрудников. Вместо этого создается таблица `Branch`, связанная с таблицей сотрудников. В таблице сотрудников указан только идентификатор филиала.

   Третья нормальная форма хороша для составления формуляров, где не нужно каждый раз заново указывать такие значения, как названия улиц или городов. Вместо этого они считываются из уже существующей таблицы и предлагаются на выбор в форме выпадающего меню. Это помогает избежать опечаток и разночтений в написании названий.

Из этого теоретического рассмотрения баз данных можно вывести следующие грубые правила.

Относитесь с подозрением к таблицам, в которых очень много граф

Часто разумнее распределить содержание по нескольким таблицам. Как новичок, вы, возможно, очень рады уже тому, что вам вообще удалось засунуть данные в таблицу, во всем последующем вам видится смутная опасность. Но в долгосрочном периоде вы тем самым окажете себе недобрую услугу.

Каждая графа должна служить одной-единственной цели

Если вы, руководствуясь содержанием других граф таблицы, используете какую-либо графу для разных типов данных, лучше разбейте эту графу на две новые.

Избегайте избыточности данных

Сохраняя одни и те же данные в нескольких разных местах, вы повышаете риск нарушения целостности базы данных. Это касается также тех данных, которые неидентичны, но тесно связаны и могут быть получены одни из других, как, например, вес брутто и нетто.

Что можно сделать еще

С одной стороны, проводя рефакторинг, вы не хотите распылять свои силы на мелочи, с другой — вам постоянно бросаются в глаза новые несовершенства вашего кода. Делайте заметки. Можете использовать для этого комментарии к коду, листок, который затем приклеите возле монитора, или специально предусмотренный список в среде разработки. Например, в таких интегрированных средах разработки, как Eclipse или Visual Studio, есть подобный список задач, который представляет в удобной форме все комментарии, снабженные определенными ключевыми словами вроде `FIX`, `HACK` или `TODO` (рис. 15.2).

Изменения, объем которых действительно невелик, вы можете выполнить мимоходом. Но программу все же необходимо протестировать: легко оказаться в ситуации, когда небольшие изменения, сделанные походя, приводят к полному отказу программы.

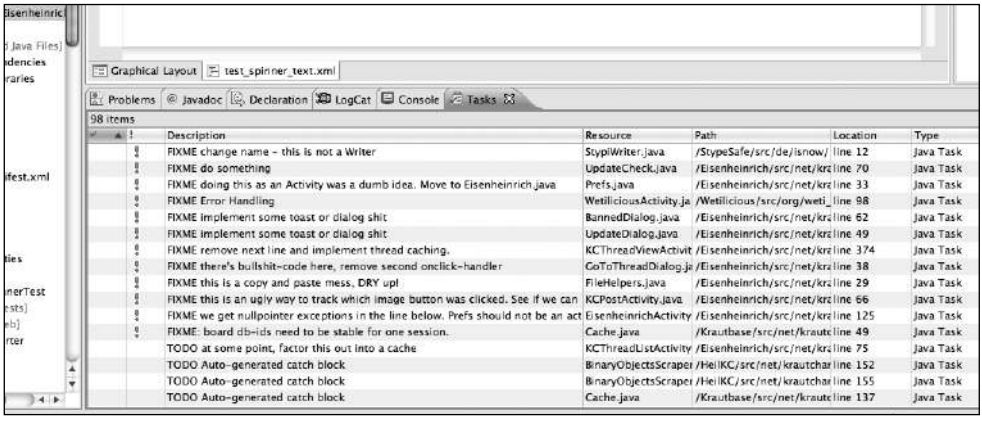


Рис. 15.2. Список задач в среде разработки Eclipse

Переименовать функции, методы, программные модули или переменные

Чистые переименования лежат скорее на границе той области изменений, которую можно назвать рефакторингом. Мы упоминаем их здесь, потому что они хорошо согласуются с целями рефакторинга — делают код понятным и удобным для работы с ним в будущем.

Чаще всего функции рано или поздно изменяют свою основную задачу. Так, в процессе разработки может оказаться, что ваша функция `printUserData()` больше не занимается непосредственным выводом символьной строки на экран, а вместо этого возвращает строку, которую затем или записывает в файл, или

отображает в окне. (Тут интересующиеся историческими вопросами читатели могут почерпнуть сведения о том, что уже само использование команды `print` для вывода информации на экран — анахронизм и эта команда сама служит примером описываемого феномена.) В этом случае своевременным решением было бы переименовать функцию в `serializeUserData()`, чтобы имя и задача вновь совпали.

Еще чаще переменные больше не содержат того, что указано в имени. Сначала переменная `$position`, возможно, еще и содержит указание относительной позиции, но потом вы обнаруживаете, что абсолютное позиционирование все же проще. В таком случае стоит переименовать переменную в `$absolutePosition`. Более подробную информацию о переименованиях вы найдете в разделе «Сначала Византий, потом Константинополь, теперь Стамбул» главы 5.

Удаляйте неиспользуемые переменные

Неиспользуемые переменные не стоит вычесывать из исходного кода, как вшей, — они не вредят работе программы. Однако они сбивают с толку читателей кода, поэтому, если уж вы о них споткнулись, стоит их удалить. Допустим, вы хотите расширить модуль и обнаруживаете переменную, функцию или класс, которые выглядят неиспользуемыми. Даже если вы не применяете вообще никакой среды разработки, в абсолютно любом редакторе вы все равно можете запустить поиск по всем файлам проекта и выяснить, действительно ли эта подозрительная штука не используется. Если это так, безжалостно удаляйте ее. Если вы не уверены в правильности такого шага, то хотя бы закомментируйте ее и укажите дату изъятия из использования. Когда вы наткнетесь на нее в следующий раз, то сможете удалить, если за это время не случится ничего плохого.

Переформатирование

Иногда за счет более подробного форматирования можно достичь лучшей удобочитаемости, как в следующем примере:

```
if (some condition &&
    some condition &&
    some condition &&
    some condition)
    do something
```

Впрочем, было бы лучше вынести сложное условие в функцию, как это описано ранее в примере с `cuteHedgehog`.

Стало ли теперь действительно лучше?

Если после всех этих перестроек вы смотрите на код, который хоть и состоит теперь только из коротких кусков, но зато содержит гораздо больше методов, классов и файлов, и если теперь вы спрашиваете себя, стал ли он действительно более удобочитаемым, чем раньше, то знайте: на эту тему спорят и специалисты. Чаще всего

вы можете сделать только что-то одно: уменьшить или количество отдельных частей, или их длину. «Что для одного проще, то для другого сложнее» — так однажды написал об этом в своем блоге Майк Тейлор[1].

В спорных случаях ориентируйтесь не на длину, а на то, какой вариант кода вы лучше поймете при следующем прочтении. Если вы работаете с другими людьми, то дело обстоит сложнее, так как вам придется выяснить, какой вариант понятнее для этих других людей. Результат всегда будет компромиссом (см. главу 18). Если у вас мало опыта или нет выраженного предпочтения ни одного из вариантов, то в спорном случае лучше уж сделайте ошибку, разбив код на слишком маленькие куски. Начинающих программистов гораздо чаще выдает слишком длинный и неструктурированный код, чем слишком большое количество отдельных частей. Мыслительные усилия, затраченные на разбивку кода на более мелкие куски, позже в любом случае окупятся.

Теперь самое важное уже сделано. Не поддавайтесь искушению продолжить шлифовать код — ваша задача состоит в том, чтобы писать программы, которые делают определенные вещи, а не в том, чтобы брать первые места на конкурсах красоты исходного кода.

В «Совершенном коде» Стив Макконнелл приводит полезное правило: «Тратьте ваше время на те 20 % рефакторинга, которые приносят вам 80 % прибыли». Вы не сразу поймете, по каким признакам распознать эти магические 20 %. С течением времени у вас разовьется чутье относительно того, когда вам стоит делать рефакторинг, а когда — скорее нет. А пока что вам достаточно помнить о том, что даже профи пренебрегают 80 % теоретически возможного рефакторинга.

Когда от рефакторинга лучше отказаться

Даже если вы программируете не за деньги, то обычно все равно не делаете это бесцельно, а хотите создать что-то определенное. Только в том случае, если вы программируете исключительно ради развлечения или чтобы испытать новый язык, решение провести рефакторинг не чревато никакими рисками и побочными эффектами. В любом другом случае рефакторинг влечет за собой определенные издержки.

Если вы хотите только переименовать функции или переменные или привести отступы в соответствие с локальным руководством по стилю оформления, то лучше оставьте все как есть. Вы не привнесете в код структурных улучшений, а в будущем и вам и другим, возможно, будет трудно понять смысл сделанных изменений. В условиях нехватки времени весьма неприятно находить в истории системы контроля версий многочисленные изменения, которые приходится рассортировывать на реальный рефакторинг и чисто косметические изменения.

Появление новых технологий также не является достаточным аргументом для проведения рефакторинга. Если вы не можете привести исключительно надежных аргументов в пользу того, что благодаря применению новой техники или нового

---

[1] Этот пост — хорошая отправная точка для дальнейшего чтения, если вы хотите узнать больше о его размышлениях на эту тему: reprog.wordpress.com/2010/03/28/what-is-simplicity-in-programming/.

средства разработки код станет значительно быстрее, полезнее или удобнее для обслуживания, то лучше оставьте все как есть. Найдите себе маленький частный проект, в рамках которого сможете наиграться с привлекательным новшеством.

В пользу того, чтобы ничего не делать, особенно многое говорит тот случай, если вы работаете с другими людьми. В командном проекте желание провести рефакторинг часто возникает по причинам, имеющим мало отношения к исходному коду как таковому. Это, например, хвастовство, прокрастинация или интрижки, направленные на получение большей власти.

Тот, кто переписывает код, написанный другим, тем самым утверждает, что он лучше понял задачу, чем изначальный разработчик. Это не проблема, если существует фактическая и всеми признаваемая разница в компетентности. Но если члены команды имеют примерно одинаковый опыт, то рефакторинг чужого кода воспринимается как не очень приятное послание. Для сохранения общественного спокойствия лучше соблюдайте осторожность и занимайтесь собственным кодом. Конечно, в командах разработчиков должно действовать золотое правило «Код — это не частная собственность» и каждый должен иметь право расширять и исправлять его. Но, несмотря на это, мы рекомендуем как можно раньше обсудить возможность глубокого рефакторинга со всеми причастными к коду людьми.

Перестраивая в ходе безудержной рефакторинговой оргии всю структуру проекта, вы приводите в смятение своих соратников. Возможно, вы лишь нарушите привычный порядок их работы, но вполне вероятно, что они почувствуют себя программистами второго сорта.

Подумайте также о том, что, преобразуя имеющийся исходный код, вы не занимаетесь чем-то таким, чем вместо вас мечтали бы заняться остальные. Вы выполняете неприятные задачи, которые ваши коллеги берут на себя так же неохотно, как и вы. У каждого бывают непродуктивные фазы, когда реализация новых программных возможностей идет очень туго. Иногда эти фазы можно использовать для сопровождения кода, но не следует из чистейшей прокрастинации переписывать код вдоль и поперек. Хотя после этого и возвращаешься домой с таким ощущением, как будто сделал что-то полезное, но общая продуктивность команды от этого падает. И даже если вы программируете исключительно для себя, не поддавайтесь искушению заняться украшательством своего кода, чтобы уклониться от трудных задач. Это то же самое, что затеять генеральную уборку квартиры в тот момент, когда нужно писать дипломную работу.

Даже если все это не про вас и вас сподвигли на рефакторинг только самые благородные цели, знайте: вас не всегда поблагодарят за рефакторинг. Вы потратите много времени на изменения, в результате которых код продолжит делать то же самое, что и раньше. Если все пойдет хорошо, не ждите бурных аплодисментов. А вероятно, что все пойдет нехорошо, — как часто вам уже доводилось изменять код, не внося в него новых ошибок? В худшем случае вы по незначительной причине разрушите что-то важное. Тем самым вы тут же навлечете на себя справедливый гнев братьев-программистов, так как рефакторинг не является самоцелью, а должен лишь упростить будущую работу. Если вместо этого он усложняет текущую работу, значит, вы что-то делаете не так.

Джеральду Вайнбергу принадлежит следующий афоризм: «Не существует столь большого, запутанного или сложного кода, который не смогло бы ухудшить сопровождение». Он предупреждает о возможности испортить сложный код сопроводительными работами, выполняемыми с благими намерениями. Итак, не занимайтесь рефакторингом кода, который вы не понимаете, потому что иначе с вами произойдет то же, что произошло с одним из авторов в случае, описанном в следующей врезке.

Код это или мусор?

«В одном из своих первых проектов я работал с ведущим разработчиком над одним Java-приложением. Одна часть написанного им кода показалась мне слишком запутанной и непонятной, так что я переделал исходный код с целью удалить избыточные места и лучше приспособить его под выполнение конкретных задач.

На следующий день он вызвал меня к себе и с некоторым недовольством спросил, зачем я искромсал на куски написанную им машину состояний. Выяснилось, что я не распознал ее как таковую и не знал, что он предусмотрел ее для конкретных расширений. Хотя сделанные мной изменения и можно было быстро откатить с помощью системы контроля версий, момент, в который мне пришлось признать, что я неправильно понял код, был не самым приятным.

Йоханнес

МАШИНА СОСТОЯНИЙ

Это парадигма программирования, в рамках которой особенно важны определенные состояния и переходы между ними. Например, необходимо удостовериться, что финансовая операция либо завершилась успешно, либо в случае ошибки была полностью отменена.

Это был очень краткий обзор темы, которой посвящены целые книги. Если после прочтения вы получили представление о том, при каких обстоятельствах рефакторинг является хорошей идеей и как он в общих чертах производится, то цель этой главы выполнена.

Если вы хотите узнать немного больше по рассмотренной теме, рекомендуем вам очень дружественные по отношению к дилетантам советы по поводу рефакторинга, содержащиеся в книге Дастина Босуэлла и Тревора Фаучера «Читаемый код, или Программирование как искусство». «Совершенный код» Стива Макконнелла также содержит подробные и понятные примеры. Желающим погрузиться в тему основательно горячо рекомендуем книгу Мартина Фаулера «Рефакторинг. Улучшение существующего кода» (Addison Wesley Professional, 1999). Книга ориентирована на продвинутых читателей и содержит несколько непростых идей, но по большей части советы Фаулера будут хорошо понятны и начинающим. Краткое изложение всех техник улучшения кода, представленных в его книге, можно найти по адресу martinfowler.com/refactoring/catalog/.

Проблема и ее решение

Катрин: «Что делать, если ты уже разобрался в проблеме и осознал весь ужас былых деяний, но та часть, которую нужно изменить, засела слишком глубоко в основании кода? Нумерация статей блога Riesenmaschine — это временная метка, которая изменяется всякий раз, когда статья подвергается внутренней обработке, и только при активации устанавливается на длительный срок. Это непостоянство внутренних номеров статей влечет за собой многочисленные неприятности. Как решал бы более опытный программист эту проблему с самого начала? И как произвести рефакторинг, если в основании кода нужно заменить один из самых глубоко лежащих камней?»

Йоханнес: «Лежащая в основании этой ситуации проблема заключается в том, что идентификаторы замахиваются на слишком многое: они пытаются быть, во-первых, временной меткой последнего изменения, во-вторых, индивидуальным идентификационным номером, а в-третьих, еще и чем-то вроде отслеживания сеанса. То есть статьи, которые редактируются в данный момент, должны снабжаться предупреждением в случае, если их захочет отредактировать кто-то еще, или как? Как раз в случае с идентификаторами часто забывают, что их важная функция — быть однозначными определителями чего-либо, которые невозможно перепутать. Они явным образом призывают подумать следующее: “Ах, тут всего одна строка, ведь я могу вписать туда что-то разумное!” Но идентификаторы с двойным значением (временные метки или адреса электронной почты) являются часто встречающейся ошибкой, которая снова и снова приводит к неприятностям».

Эту перегруженность переменной можно было бы устранить, если бы при создании статьи ей было присвоено в качестве индивидуального идентификационного номера большое случайное число, которое больше бы никогда не изменялось. Другие функции можно было бы реализовать в новых полях на XML, например `timeLastEdited` и `lastEditor`. Теперь эти поля можно было бы в любое время изменить, не трогая индивидуальный идентификационный номер.

Часто проблему не удается решить с ходу, тогда нужно разложить ее на части, а потом по очереди позаботиться о каждой из частей. В таком случае пишутся так называемые функции с бутылочным горлышком — функции, которые имеют монополию на управление определенными данными. В приведенном ранее случае это были бы, например, функции `getPostingID()`, `setPostingID()` и `updatePostingID()`.

Далее эти функции сперва продолжили бы производить нумерацию статей неудачно выбранным способом, однако централизованно — ни в каком другом месте программы идентификаторы не вычислялись бы и не изменялись. В логике нумерации ничего бы не изменилось, поэтому обычно такую перестройку можно реализовать шаг за шагом, не разрушая построенный на всем этом карточный домик: там, где раньше изменялись идентификаторы, происходит обращение к функции с бутылочным горлышком `updatePostingID()`. Поскольку в функции содержится прежняя логика, другие, еще не обработанные части кода могут продолжать возню с идентификаторами на свой манер. Аналогичным образом, через обращение к функции `sortArticle()`, заменяются места кода, которые занимались сортировкой статей.

Когда все это сделано и хорошо протестировано, большая часть проблемы уже решена. Теперь можно остановить систему, чтобы получить возможность спокойно изменить логику вычисления идентификаторов. Это делается только централизованно, с использованием небольшого количества функций с бутылочным горлышком.

Правда, в конце возникает еще одна проблема — проблема миграции данных: старый способ нумерации статей не подходит к новому, так как старые идентификаторы состояли из временных меток, а это хоть и не расположенные по порядку, однако постоянно возрастающие цифры. Новые идентификаторы, напротив, являются случайными числами. С помощью старых идентификаторов статьи можно было сортировать, а с новыми это невозможно. Поэтому нужно распределить обе функции прежних идентификаторов, а именно функцию индивидуального идентификационного номера и функцию помощника при сортировке, в два поля (`ID` и `timeLastEdited`) в файловом формате. К сожалению, это приведет к тому, что старый и новый файловые форматы больше не будут совместимы.

Чтобы не приходилось иметь дело с двумя форматами, имело бы смысл написать небольшую программку, которая преобразовывала бы старые статьи в новый формат: брала бы старый файл статьи, считывала значение старого идентификатора и записывала бы это значение в `timeLastEdited` и новый идентификатор. Эту программу нужно прогнать один раз по всем статьям, после чего можно будет редактировать старые статьи, и это не будет каждый раз приводить к изменению их идентификаторов. Поле `timeLastEdited`, напротив, будет изменяться при каждом редактировании.

До рефакторинга и обновления данных массив данных выглядел бы так (без XML-тегов):

```
Article 1236739535
    content blafasel
```

После рефакторинга и обновления данных массив данных выглядел бы следующим образом (снова без XML-тегов).

Старая статья:

```
Article 1236739535
    lastTimeEdited 1236743135
    lastEditor null (так как в старом формате этого поля не было)
    content blafasel
```

Новая статья, созданная в новом формате:

```
Article 2345903094024305928348902 9435
    lastTimeEdited 1486587262
    lastEditor Johannes
    content blafasel
```

16 Тестирование

> Процесс тестирования — это бесконечное сравнение невидимого с неоднозначным с целью того, чтобы избежать немыслимого в отношении к неизвестному.
>
> *Джеймс Бах*

Программное обеспечение можно тестировать различными способами. Для начала программу запускают, чтобы убедиться в корректности ее функционирования. Затем включают тестовые программы, которые проверяют исходящий из системы контроля версиями код, компилируют его с образцом и тестируют. Существуют тесты на нахождение ошибок в продолжительности обработки данных (которые могут привести к сбою в программе), ошибок, допущенных в процессе написания алгоритма, тесты для диагностики памяти, проблемы с которой могут привести к поглощению программой все большего объема памяти, программы, определяющие данные, искаженные в процессе трансформации, и сертифицирующие тесты, которым подвергается ПО, чтобы быть задействованным в определенных отраслях.

Для чего нужно тестирование

Дело не только в лени, которая мешает нам прилежно писать тестовые сценарии, но и в постоянном ощущении нехватки времени. Отчасти это верно, ведь тесты, впрочем, как и другие способы обеспечения качества (например, ведение документации), тормозят разработку ПО. Однако существует ряд причин, по которым стоит тестировать как минимум наиболее важные части приложения.

В отличие от прочих форм документации, в тестовых сценариях прописывают изначально заложенные в коде параметры. Логическая схема программы, прописываемая в комментариях, не всегда будет изменяться в соответствии с преобразованным кодом. Не стоит также изменять параметры кода, адаптируя их под тестовые сценарии, — это может привести к ошибкам.

Тесты являются хорошим основанием для реорганизации кода. Если после многократных изменений структуры программы запустить тестовый сценарий, он укажет на некорректно работающий код. Обычно это труднораспознаваемые или же устаревшие версии кода — именно для таких версий тестирование является наиболее важным. Таким образом, плохие результаты тестирования могут сыграть вам на руку — они указывают на проблемные места в программе.

Когда вы работаете в команде, тесты являются важным компонентом совместной работы: если все состыковки между программными модулями успешно прошли тестирование, разработчики могут быть уверены в том, что всегда смогут перепроверить их таким же образом. И наоборот, при выявлении новых ошибок можно тестировать и чужой код, а затем сравнивать результаты, чтобы установить, в чем заключается ошибка.

Способы тестирования

Существует множество способов тестирования, не конкурирующих между собой и различающихся лишь сферами использования.

Системное тестирование

Даже являясь не самым лучшим программистом, вы бы, вероятно, выбрали этот тип тестов. Он заключается в том, что вы то и дело щелкаете на настройках и функциях программы, чтобы убедиться в ее исправности.

Не стоит пренебрегать такими тестами, так как зачастую с их помощью можно выявить определенные проблемы. Например, научная обработка данных могла бы включать в себя следующие этапы.

- Считывание данных.
- Упорядочение всех точек данных. Для этого программа находит точку с высоким значением и делит все последующие значения на этот максимальный показатель. Таким образом точки упорядочиваются в интервале от –1 до 1.
- Статистический анализ упорядоченных ранее точек.

Если в нашем примере обработка данных является частью веб-приложения или программы с пользовательским интерфейсом, то отслеживать этапы и руководить ими можно через пользовательский интерфейс. Пользователь выбирает входной файл, система обрабатывает его и сохраняет в указанном месте. Взаимодействие пользователя с системой трудно симулировать с помощью автоматизированных тестов. Хотя и существуют программы для записи макросов, например Selenium, которые могут записывать и воспроизводить действия пользователя, в большинстве случаев это не самый удобный вариант.

Поэтому при маленьких и среднего масштаба проектах не прибегают к автоматизированному системному тестированию и проводят его вручную. Обычно сам разработчик или же бедный практикант садится за компьютер и проверяет программу в режиме пользователя, то есть действуя как среднестатистический пользователь. Так могут всплыть некоторые ошибки, например, если при нажатии на определенную кнопку происходит неверное действие.

Типичные проблемы при использовании этого типа системного тестирования будут описаны далее.

Каждый пользователь применяет лишь малую часть функций программы. Являясь рабами своих привычек, мы обычно работаем со знакомым набором функций. Щелкаем на привычных ярлыках и кнопках, думая: «Сейчас заканчиваю работу

над последним окном, как обычно, представлю релизную версию клиентам и наконец-то домой!» К сожалению, пользовательские интерфейсы редко являются линеарными, поэтому, если кто-то из ваших коллег изменит боковую ветвь древовидной структуры или значение какой-либо переменной, вряд ли вы это заметите.

Если до этого вы сто раз тестировали программу, то, скорее всего, на сто первый раз будете в полусонном состоянии щелкать на всплывающих окнах, вместо того чтобы внимательно высматривать ошибки и изменения. Если значения изменятся и не будут соответствовать исходным, вряд ли вы это заметите, так как ваш взгляд будет прикован лишь к кнопке OK. Опыт показывает, что таким тестированием можно найти лишь очевидные и грубые ошибки.

Обычно мы пишем тестовые сценарии для одних и тех же ситуаций, при этом часто упускаем из виду, что следует проверять все аспекты программы, в том числе нетипичные случаи. Например, вам нужно войти в приложение. Если вы являетесь разработчиком или тестировщиком, то, скорее всего, ваши данные будут первыми введенными в систему. Однако может случиться, что при регистрации пользователей произойдет сбой из-за логина, набранного китайскими иероглифами. В код может закрасться ошибка, которая будет мешать регистрации пользователей. В рамках модульного тестирования гораздо легче переписать отдельную схему баз данных, при ручном же тестировании это невозможно.

Ручное тестирование занимает много сил и времени. Поэтому либо его проводят неохотно, либо на определенной стадии развития проекта вообще не проводят. В конце концов, разработчикам платят не за то, чтобы они целыми днями щелкали на кнопках, а за написание программ. По этим причинам мы советуем меньше времени уделять ручному тестированию и как можно чаще прибегать к автоматизированным тестам. Но если вы все равно решили тестировать программу вручную, то рекомендуем придерживаться следующих простых правил.

- **Составьте план работы.** Если расписать, какие действия и в какой последовательности следует тестировать, можно целенаправленно задавать различные параметры для программы, проверяя таким образом разные случаи в работе программы. Задание можно разделить на всех участников вашей команды, что позволит избежать субъективности в тестировании. И чтобы не допустить ошибок по невнимательности, составьте контрольный список, вы сможете распечатать его и помечать выполненные задания.
- **Тестируйте избирательно.** Не пытайтесь сделать все и сразу. Если во время разработки вы будете помечать, какой именно код изменяли, то сможете сэкономить время на полном тестировании. Лучше потратьте это время на тщательное тестирование измененных параметров программы. Если же вы будете проводить полное тестирование, то начинайте с самого начала — с регистрации или же заполнения базы данных.
- **Используйте скриншоты.** Делайте скриншоты, если надо заполнить много полей различными данными. Так вы впоследствии сможете сравнить скриншот с тем, что у вас на экране.
- **Тестируйте на ошибках.** Придумайте парочку изначально неверных данных и в рамках тестирования вводите их в систему. Наблюдайте за реакцией программы: она может дать сбой, никак не среагировать или же выдаст четкое и понятное даже обычному пользователю сообщение об ошибке.

Модульное тестирование

Модульные тесты — это программы, которые загружают определенные модули, а также функции и части вашего ПО с различными вводными данными и сопоставляют результаты с заданными значениями. Этот способ тестирования по праву является самым рекомендуемым как наиболее эффективный, быстрый и легко поддающийся автоматизации.

Модульные тесты также являются частью документации. В тестовых случаях сохраняются все условия тестирования, таким образом можно узнать, какие выводные данные получатся при определенных вводных данных. Модульные тесты — это в некотором роде выполняемые комментарии.

С помощью модульных тестов хорошо тестировать и параметры безопасности. Вы можете написать тест, который будет симулировать вводимые пользователем данные с внедрением XSS- и XQL-кода, и тем самым проверить, будет ли программа их распознавать.

Модульные тесты являются отрадой для лентяев. Ведь всего-то надо задать конкретные тестовые параметры и после сравнить их с прописанными в коде данными.

Например, вы написали функцию `readExperiments ()`, которая при нескольких экспериментах считывает с CVS-данных (значения, разделенные запятыми) все измерительные данные. В таком случае вы пишете тест в соответствии с процессом выполнения программы внутри этой функции. Вероятно, функция может повести себя следующим образом.

- Откроет данные.
- Будет построчно считывать измерительные значения и для каждой строчки задавать новый эксперимент в виде объекта или массива.
- Разобьет каждую строку на отдельные измерительные значения и впишет туда эксперимент (массив).
- Составит список экспериментов в соответствии с числом строчек.
- Отобразит список экспериментов.

Каждый из этих этапов можно проверить с помощью модульного тестирования, но если у вас мало времени или вы еще недостаточно хорошо освоили этот тип тестирования, можете для начала написать пару тестов, которые будут проверять функцию как целое.

Для этого приложите к данным с исходным кодом входящие данные, которые должны содержать правильные значения, и можно приступать:

```
function testReadExperimentsIsNumExperimentsOK () {
   var experimentList = readExperiments("/home/johannes/sampledata_20130312.
csv");
    assert(experimentList.length == 5);
}
```

Поздравляем! Вы написали свой первый модульный тест!

Утверждения

Если вы не владеете хорошо командой `assert()` из приведенного ранее примера, можете найти больше информации по данной теме в главе 26. Однако общий принцип

довольно прост: команда условия типа Boolean, и если результат будет `false`, то программа прерывается из-за ошибки или же `assert()` запускает процесс обработки исключений. В модульном тестировании команда `assert()` является ключевым критерием эффективности или неэффективности теста. Она делает ту работу, которую вы выполняете при ручном тестировании.

В зависимости от языка и системы отладки команда может называться поразному. Например, в Angular.js, фреймворке JavaScript, это выглядело бы следующим образом:

```
expect(experimentList.length).toBe(5);
```

Модульное тестирование зарекомендовало себя как площадка, где команды соревнуются в написании утверждений, наиболее приближенных к структуре человеческого языка. Это приводит к утверждениям типа:

```
expect(experiment.name).to.be.a('string');
```

Тест можно считать успешным, если в переменной `experiment.name` находится какая-нибудь строковая переменная.

В другом варианте вместо `expect` или `assert` используется `should`:

```
experimentList.should.have.length(5);
```

Данный код проверяет, будет ли массив в конце выполнения программы содержать пять элементов.

Тестовый набор (Test Suite)

Модульный тест, приведенный ранее, проверяет лишь один аспект функции `ReadExperiments()`, а именно, найдет ли функция правильное количество экспериментов во вводимых данных. Если найдется всего четыре — шесть, тест выдаст ошибку. Однако тест не может проверить, содержат ли эксперименты рациональные значения.

При выполнении следующего теста функция пройдется по списку экспериментов и проверит каждый эксперимент на правильность значения:

```
function testReadExperimentsNamesOK () {
    var names = ['exp_ 1356346800', 'exp_ 1356347203', 'exp_ 1356350314',
'exp_1356351341', 'exp_ 1356354958'];
    var experimentList =
readExperiments("testdata/sampledata_20130312.csv");
    for (var i = 0; i < experimentList.length; i++) {
        assert(experimentList[i].name == names[i]);
    }
}
```

Проводя этот тест, вам не нужно было бы беспокоиться, все ли эксперименты считываются, так как вы уже проверили это с помощью прошлого теста. Модульное тестирование можно сравнить со стеной. Когда каменщик строит стену, он кладет кирпич за кирпичом, начиная снизу и заканчивая на самом верху. Сначала можно тестировать отдельные мелкие части программы и плавно переходить к тестированию программы в целом. Это удобно, так как тесты находятся на виду и легко

сфокусироваться на определенной проблеме, поскольку каждый тест проверяет только один аспект программы.

Так, из ряда отдельных тестов можно сформировать тестовый набор, который будет состоять сначала из совсем простых тестов, постепенно дойдет до более сложных и со временем станет таким объемным, что сможет проверять все аспекты модуля программы за раз.

Фикстуры (Fixture)

Зачастую невозможно протестировать код в полной изоляции. К примеру, ваш код должен обращаться к базе данных, чтобы считать или сбросить значения. В этом случае вам надо для начала создать тестовую базу данных и написать дополнительный код, который будет заполнять нужные таблицы значениями, а после теста удалять их.

Для таких случаев прочно укоренилось понятие *«фикстура»*. Она описывает среду, в которой проводится тест. Вдобавок к собственно тестовой фазе идут модули тестирования setup и teardown. На этапе setup налаживается связь между базами данных, заполняются таблицы вводных данных и удаляются выводные данные из таблиц. После проведения теста и проверки результатов, то есть на этапе teardown, наоборот, разрываются связи между базами данных или же удаляются временные данные. Из-за модулей setup и teardown отдельные тесты не завершаются до конца, что зачастую и невозможно из-за выполнения программы. Если вашей программе надо сначала соединиться с удаленным сервером или же считать базу данных, то стоит это делать один раз за все время тестового запуска, а не с запуском каждого отдельного теста.

Для исправного модульного тестирования важными являются следующие факторы.

- **Способность оценивать проделанную работу.** Каждый тест должен уметь определить, насколько хорошо он справился с задачей. Тест, преобразующий вводные данные в выводные, которые впоследствии проверяет человек, не является модульным.
- **Независимость.** Отдельные программы тестового набора должны быть независимы друг от друга. Хорошо, когда есть возможность запустить любой тест в любое время. Это нужно для того, чтобы вы могли быстро получить общее представление о возникшей неполадке. Если по предварительным условиям тестовой программе нужна заполненная база данных, то она должна (насколько позволяют условия выполнения) об этом позаботиться сама. Хорошо, если нет необходимости проводить сначала тест A, преобразующий данные в определенную структуру, а после — тест B, который будет основываться на этих структурах.
- **Соотнесение с кодом.** Тесты должны быть сгруппированы в зависимости от кода, который они тестируют. Значит, все тесты с функцией `readExperiments()` должны находиться в файле, который мог бы носить название `TestReadExperiments`. Желательно, чтобы в этом файле не располагались другие программные модули, и, наоборот, те, которые связаны с `readExperiments()`, не стоит разбрасывать по разным местам.

- **Завершенность.** Каждый тест должен быть завершенным, то есть не запрашивать данные ввода и не производить выводные данные. Если тест выполняется нормально, то ничего происходить не должно. При сбое теста должно отображаться сообщение об ошибке, например, как при обработке исключений, — дальнейшие действия зависят непосредственно от языка программирования.
- **Автоматизированность.** Тесты должны проходить автоматически. Это экономит время и силы и помогает быстро выявить ошибку. Например, если после процесса отладки и реструктуризации вы сможете быстро провести ряд тестов, то у вас будет возможность убедиться, что в измененный код не закралась ошибка.
- **Тестирование на ошибках.** Кроме описанных тестов, которые проверяют работу программы с правильными данными, следует писать тесты, которые будут задавать ошибочные или предельные значения, например через раз `NaN` или `null` вместо числовых значений, CSV без запятых, знаки в кодировке UTF-8, неправильно оформленные вводные данные и т. п. Хорошо написанная программа при неверных вводных данных должна высвечивать ошибку, а не воспроизводить строку базы данных наполовину.
- **Сохранение тестовых данных.** Тестовые данные относятся к программе и поэтому должны быть прописаны вместе с кодом в системе управления версиями. Если же данные конфиденциальны или слишком больших размеров, то, скорее всего, это невозможно. В качестве альтернативы можно пометить себе, где находились данные и как они назывались. Быть может, вам повезет и через пару лет они еще сохранятся.
- **Библиотеки тестирования.** Для многих языков существуют тестовые фреймворки, например jUnit (Java), NUnit (.NET), Test::Unit (Ruby), Mocha (JavaScript) или unittest (Python). Они способствуют разделению модулей тестирования setup, test-suite, teardown, а также оптимизируют процесс выполнения функции `assert()`.

  Это маленькие фреймворки, которые имеют ограниченный набор функций. Многое вы могли бы написать и сами, но библиотеки тестирования экономят усилия, так как при их использовании не нужно подтверждать уже известный результат.

Валидация данных

В финансовой и научной сферах особенно часто проводятся операции с большим объемом данных, которые трансформируются и распределяются по базам данных или переносятся из одной базы данных в другую, созданную по другой схеме. Поэтому зачастую важна не столько проверка кодов, сколько проверка на целостность больших объемов данных. Даже если вы импортируете всего одну клиентскую базу данных из Excel в базы данных СУБД, то не сможете вручную проверить каждый импортированный файл. В таких случаях погрешность вычислений, логические ошибки в процессе преобразования данных и другие неточности представляют собой большую опасность, так как не приводят к сбою системы и поэтому могут долго оставаться незамеченными и в то же время наносить огромный вред. Следующие методы помогут вам предугадать подобные случаи.

Обзорная проверка всего участка данных

Если вы, например, знаете, что на импортируемом участке данных записи разделяются между собой пустыми строчкам, то в любом текстовом редакторе можно подсчитать, сколько пустых строк было на этом участке данных. Таким образом, после импорта вы сможете сосчитать, сколько новых записей появилось в вашей базе данных. Если числа строк не совпадают, следует искать ошибку. Если вы импортируете клиентскую базу, то с помощью любой программы обработки электронных таблиц можно узнать, сколько записей содержится в каждом типе клиентских баз.

При импорте данных в базу данных для облегчения работы можно использовать ограничения. Для этого нужно прописать во всех столбцах ограничение `NOT NULL`, таким образом, что, если в перемещаемых данных будет отсутствовать поле, которое не подпадает под ограничение, произойдет сбой. Если вы одновременно переносите данные в несколько таблиц, есть смысл использовать *внешние ключи (Foreign Key)*, чтобы данные импортировались корректно. Для начала создайте тестовую базу данных и переносите данные туда, если вы не уверены, что в случае сбоя программа сможет восстановить процесс импорта данных.

ВНЕШНИЕ КЛЮЧИ

Если надо соотнести данные из таблицы A с данными из таблицы B, то таблица B должна включать внешние ключи, которые будут соответствовать первичным ключам в таблице A. Ограничения отбирают в таблицу B лишь те данные, внешние ключи которых соответствуют первичным.

Выборочный контроль

Выбирайте отдельные участки данных и вручную проверяйте, насколько правильно они были считаны или импортированы. Стоит заметить, что это весьма трудоемкий процесс, который не гарантирует стопроцентную правильность.

Обратная обработка

Напишите программу, которая будет экспортировать данные обратно в изначально заданный формат, и таким образом проверяйте соответствие импортированных данных экспортированным. К сожалению, этот метод не всегда возможно использовать, например, если для заполнения пробелов вы импортируете из нескольких источников. Создание экспортирующей программы может занять столько же времени, сколько и написание программы для обратного процесса.

Проверка достоверности

Если в таблице есть графа с процентными величинами, следует добавить ячейку, в которой будут суммироваться данные из этой графы. Если сумма будет превышать 100 %, значит, где-то вкралась ошибка.

Даже если программы обработки электронных таблиц не пользуются особой популярностью у разработчиков, они являются хорошим способом проверки данных. Щелкая на основаниях граф, можно быстро настроить фильтры. Таким образом, большая часть ненужных данных отображаться не будет. Вы можете

сортировать участки данных по графам. Можно также использовать функции суммирования и обусловленное закрашивание ячеек (например, если сумма в графе C меньше, чем в ячейке D10, ячейка будет закрашена в красный цвет). Это помогает быстро различить типы данных и их целостность.

Тестирование производительности

Тестирование производительности, в отличие от модульных тестов, не выдает сообщений о корректной или некорректной работе программы. Оно показывает продолжительность выполнения функции или же количество обработанных данных в секунду. Результаты не являются абсолютными величинами и должны анализироваться с учетом показателей прошлых тестов.

Однако не стоит слишком много времени проводить за анализом производительности, так как, скорее всего, вы пишете программу, рассчитанную на тысячи пользователей: чем меньше пользователей пользуются программой, тем меньшему количеству людей будет важен уровень производительности. Если со временем вы разработаете версию программы, которая будет пользоваться большой популярностью, то, вероятно, ваш доход позволит вам уделить время оптимизации приложения. Если речь идет о бесплатной программе, то, возможно, найдутся единомышленники, которые смогут взять часть работы на себя.

Имея дело с серьезной ошибкой производительности, следует как можно раньше начинать с тестирования, чтобы следить за динамикой работы программы. Хорошим примером может послужить Mozilla. В браузере Firefox долгое время возникала проблема, связанная со скоростью работы Java-кодов. Конкурентные браузеры имели более оптимизированный под среду выполнения Java-код, из-за чего Firefox по сравнению с ними считался медленным браузером. Чтобы мотивировать разработчиков и похвастаться своими успехами всему миру, разработчики Mozilla создали сайт AreWeFastYet.com, где сопоставляли скорости выполнения Java-кодов браузеров Mozilla, Google Chrom и Apple Safari и отображали показатели скорости в виде диаграммы.

Являясь разработчиком-одиночкой, вы вряд ли имеете возможность запустить такой дорогостоящий мониторинг производительности. Однако, тестируя производительность основных компонентов программы, вы сможете легко и быстро распознать проблемы. В частности, это касается мобильных приложений, так как процессоры смартфонов относительно медленные, за счет чего потребляют больше электроэнергии. Если же вы пишете веб-приложение, следует обращать внимание на время отклика сервера и время выполнения JavaScript.

Самый простой способ измерения производительности заключается в том, чтобы засечь время начала и конца работы определенного кода или кодов. Если вычесть из времени окончания работы время начала, получится время, которое компьютер тратит на выполнение определенного кода. Целесообразно вести дневник записей измерений, чтобы впоследствии можно было анализировать результаты. Если показатели будут ухудшаться, следует выяснить причину.

Несколько более сложный метод представляет собой тестирование с помощью библиотек, к примеру *jvisualvm* для Java, *timeit* для Python и *Benchmark* для Ruby.

Их также называют инструментами профилирования или просто профайлерами, так как они позволяют анализировать отдельные фрагменты, а также скорость вызова определенных функций. Если вы тестируете реляционную базу данных, воспользуйтесь функцией `explain`, которая может дать развернутую информацию о скорости выполнения запросов.

Используя тесты производительности, следует остерегаться некоторых ловушек. Например, вы могли бы использовать описанную ранее функцию обработки научных данных следующим образом:

```
function readExperiments(experimentFileName, resultFileName) {
    var startTime = time.now();
    print ("Start time: "+startTime);
    var experimentFile = open (experimentFileName);
    var experiments = new Array();
    while (experimentFile.canRead()) {
        var line = experimentFile.readLine();
        experiments.add (parseExperiment(line));
    }
    var results = processExperiments (experiments);
    var resultFile = open (resultFileName);
    resultFile.write (results);
    var endTime = time.now();
    print ("End time: "+endTime);
    print ("Total time: "+ (endTime – startTime));
}
```

Этот код хоть и задает время обработки данных, однако содержит ряд трудностей.

Изменения кода

В описанном ранее коде измерение времени выполнения программы было задано для конкретного случая. Это не помешает, если вы ищете проблему производительности, ведь вам нужны просто данные измерения. Однако не следует забывать, что любые изменения кода могут повлечь за собой ошибки, к тому же для использования программы в полноценном режиме коду все равно придется вернуть исходный вид, и измерить производительность таким способом вы больше не сможете.

Большой разброс результатов данных

Тестовый запуск выполняется всего один раз. Однако стоит проводить каждый тест по нескольку раз, так как иногда фоновая программа может загружать процессор, из-за чего тест может выполняться значительно медленнее обычного. Хоть такие перепады и случаются крайне редко, такая большая разница в продолжительности работы все же может сбить с толку. По этой причине целесообразно проводить тесты по нескольку раз и использовать при этом среднее значение результатов теста, или медиану[1].

---

[1] В зависимости от задач теста результаты могут колебаться. Медиана всегда будет отображать средние показатели теста. Если вам надо проанализировать колеблющиеся показатели, стоит рассматривать также максимальные и минимальные значения.

Измерение ненужных данных

Если вы измеряете время выполнения функции, которая считывает до 1000 точек данных с жесткого диска, то программа высчитывает среднее значение, а затем вписывает результаты в файлы данных, поэтому, скорее всего, вы измеряете не совсем то, что хотите измерить. Файловые операции чтения и записи занимают слишком большую часть от всего времени выполнения функции. Поэтому для начала стоит считать данные, затем провести тестирование, дождаться результата и только потом начать выполнять файловые операции.

Время выполнения зависит от объема записи данных

Некоторые алгоритмы имеют неизменное время выполнения вне зависимости от объема данных, однако у большинства это время увеличивается в зависимости от объема записываемых данных. Эта зависимость выражается так называемым «О большим». Если вам не повезло и вашей программе для считывания больших объемов данных нужно экспоненциально больше времени, то покупка быстрого процессора вряд ли решит проблему. Вам придется изменять алгоритм или же довольствоваться при обработке небольшими объемами данных. Чтобы оценить работу алгоритма «О большое», стоит создать тестовые данные различных размеров. Например:

```
function testProcessExperiments() {
    var experimentFile = open ("testdata/smallsampledata.csv");
    var experiments = new Array();
    while (experimentFile.canRead()) {
        var line = experimentFile.readLine();
        experiments.add (parseExperiment(line));
    }
    var startTime = time.now();
    var results = processExperiments (experiments);
    var endTime = time.now();
    return endTime – startTime;
}
function testProcess20Experiments() {
    var smallSampleTime = new Array();
    var largeSampleTime = new Array();
    for (var i = 0..20) {
        if (random() > 0.5) {
            smallSampleTime[i] = testProcessExperiments ("testdata/
            smallsampledata.csv");
        } else {
            largeSampleTime[i] = testProcessExperiments ("testdata/
            largesampledata.csv");
        }
    }
    print "Small experiment median: "+math.median(smallSampleTime);
    print "Large experiment median: "+math.median(largeSampleTime);
}
```

Описанные ранее проблемы можно решить, если:

- измерять время выполнения функции `processExperiments()` в рамках теста. Функцию `readExperiments()` трогать не стоит, так как файловые операции могут повлиять на результаты тестирования;
- обрабатывать отдельно данные больших и маленьких размеров;
- через каждые десять раз чередовать данные, чтобы не возникало непредвиденных колебаний в скорости.

В качестве дополнительной меры предосторожности следует записывать также медиану.

Правильно тестировать

Никогда не проводите тесты на рабочих данных, так как они могут изменяться, а вместе с ними и целые базы данных. Если в ходе тестирования вы используете такие данные, то рискуете получить искаженные результаты. Поэтому стоит проводить тесты на неизменных данных. Если во время дальнейшей разработки программы поменяется формат данных, следует создать новые тестовые данные и переписать тестовые сценарии. К сожалению, других способов пока не существует.

Группируйте тесты по связанным между собой функциям и объектам. Если при проверке определенного кода какой-нибудь тест обнаружит ошибку, то, скорее всего, это повлечет за собой ошибки и в других тестах. В таком случае удобно, если тесты разграничены по группам, — это облегчит последующий анализ.

Старайтесь проводить тестирование на разных объемах данных и разных значениях. Следует тестировать NULL-значения, 0, –1 и 1, так как при таких значениях чаще обнаруживаются ошибки. Также стоит регулярно анализировать процесс обнаружения и устранения ошибок. Только так вы сможете узнать, как ведет себя программа, если что-то идет неправильно. Для этого можно разрабатывать тесты с пустыми строками, указателями, NaN-значениями и другими необычными и недействительными значениями.

Не впадайте в крайности. Вы не обязаны тестировать, сохраняет ли база данных файлы или выводит их обратно. Это весьма скрупулезно проверяют создатели программы. Также вы можете не тестировать работу библиотек и фреймворков. Сконцентрируйтесь на своем коде, однако и здесь не перегибайте палку. Метод чтения (геттер), например приведенный далее, вам также можно не тестировать:

```
void setColor (newColor) {
    this.color = newColor;
}
```

У программистов критерием качества считается достижение модульными тестами 100 % покрытия кода. В теории это хорошо, однако на практике не всегда выполнимо, особенно если по профессии вы не программист-разработчик. Если вы тщательно проверите все важные функции своей программы и станете делать это регулярно, то уже будете лучше многих программистов.

17 Предупреждающие знаки

> #чегонехватает: закона, запрещающего писать предупреждение «Бутылку не класть» шрифтом меньшего размера, чем 500 пунктов.
>
> *Катрин Пассиг/@kathrinpassig, Twitter, 16 августа 2013 года*

Имея средние навыки программирования можно добиться успехов. Тот, у кого есть молоток и гвозди, но кто не имеет понятия, как ими пользоваться, может построить себе дом. Возможно, он будет немного перекошенный и не станет претендовать на архитектурные награды, однако выполнять свои основные функции он сможет. Например, укрывать своих хозяев от дождя. В строительстве дома так же, как и в программировании, о многом можно догадаться самому. Что-то приходит с опытом, например: коротко отпиленные доски не так-то просто подогнать под нужный размер, так же как и когда-то стертый код можно навсегда потерять, если заранее не позаботиться о резервной копии или системе контроля версий.

Однако бывает не так-то просто отгадать правильный порядок действий и последствия ошибок видны далеко не сразу. Снова приведем пример с постройкой дома. Тот, кто не знает, что по стандарту проводку прокладывают под штукатуркой, будет прокладывать ее так, как ему удобно. Жильцы, пришедшие после него, захотят повесить картину и попадут гвоздем в проводку. Может быть, сразу они ничего и не заподозрят, однако когда-нибудь кабель начнет дымиться и дом загорится.

В этой главе собраны несколько предупредительных знаков, которые помогут распознать проблему. Даже если вы живете под девизом «Какая разница, и так сойдет!», вы окажете себе и будущим хозяевам дома большую услугу, если хотя бы пробежитесь глазами по этой главе. Вам не надо запоминать все детали, но если когда-нибудь вы будете готовы забить гвоздь в стену, что-то вас остановит и вы вспомните, что что-то когда-то читали. В таком случае эта глава выполнит свое назначение.

GET- и POST-запросы

Область применения: веб-программирование.

Последствия неведения: Googlebot удаляет весь контент со страницы.

Краткая информация: POST-запросы используются для передачи данных серверу, например, если мы закачиваем что-нибудь на сайт или же просто отсылаем заполненный формуляр. Таким образом, на сервере может появиться что-то новое

(например, запись в блоге) или же измениться что-то уже существующее (например, профиль пользователя). POST-запросы могут запускаться с помощью кнопки Отправить или JavaScript, но не простой ссылкой.

GET-запросы используются для получения данных с сервера, например, результатов поисковых запросов. GET-запросы не должны вызывать каких-либо изменений на сервере, кроме незначительных, — запросы фиксируются в журнале сервера. Однако с помощью этих запросов можно устроить различные сбои на сервере. Например, вы пишете систему управления контентом для большого правительственного сайта, ссылки для удаления которого работают через GET-запросы. Спустя пару дней ваш красивый новый сайт посещает Googlebot, чтобы отсканировать страницы для поискового индекса Google. Согласно заданной установке он переходит по всем указанным ссылкам и за ночь удаляет весь контент[1].

GET-запросы не должны вызывать проблем или сбоев. Всегда должна быть возможность просто перезагрузить страницу, не ожидая каких-либо неприятных сюрпризов.

GET-запросы сохраняются в журнале браузера, а также в виде лог-файлов в журнале сервера. Было бы хорошо, если бы в журнале не встречались клиентские данные и номера кредитных карточек. (Вероятно, так фирма Connect 2 Cleanrooms Ltd. из Киркби Лонсдейл пользуется данными одного из соавторов этой книги. При каждом вызове сохраненной в браузере страницы этой фирмы происходило автоматическое оформление заказа и автору приходилось получать очередную громоздкую посылку из Англии.)

POST — это сообщение, которое, как посылка, может содержать в себе что-нибудь, например файл, который надо загрузить на сайт. GET в данном случае можно сравнить с телеграммой, здесь все передается через URL. У старых браузеров могут возникнуть проблемы с использованием слишком длинного (от 2000 знаков) GET-запроса.

Результат отосланного POST-запроса нельзя занести в закладки, в отличие от GET-запросов. Одно лишь это является веской причиной для того, чтобы пользоваться поиском с помощью GET-запросов, а заказывать покупки с помощью POST.

Так как URL содержит все данные GET-запроса, все параметры находятся на виду у каждого. Даже люди без особых наклонностей к хакерству иногда испытывают искушение попробовать заменить параметры запроса. (Хотя нельзя утверждать, что и POST-запросы являются безопасными. При использовании простого браузерного дополнения или же командной строки можно легко изменить параметры.) Это является одной из причин необходимости тщательно следить за безопасностью на серверах. Больше информации вы найдете в главе 25.

В качестве утешения: очень много людей часто путают GET- и POST-запросы.

Менее утешительным является то, что мы как пользователи Интернета расплачиваемся за эти ошибки и в итоге получаем два холодильника вместо заказанного одного.

---

[1] Поучительные подробности этой истории можно прочесть на сайте thedailywtf.com/articles/the_spider_of_doom.aspx (на английском языке). Использование GET- и POST-запросов было лишь одним из многих неосторожных поступков в этой истории.

Кодирование знаков

Область применения: везде.

Последствия неведения: вы попадетесь разработчику Джоэлу Спольски, который когда-то написал: «Если в 2003 году вы будете работать программистом и я поймаю вас на том, что вы не будете знать базовых вещей про знаки, наборы знаков и символов, кодировку знаков и Юникод, то в качестве наказания, клянусь, я заставлю вас шесть месяцев чистить лук на подводной лодке». Кроме того, ваши тексты могут выглядеть странно или не отображаться вовсе у пользователей из других стран, использующих другие типы шрифта или настройки браузера.

Краткая информация: «просто текста» не существует. Каждый текст написан в определенной системе кодирования знаков (по-английски — character encoding, или просто Encoding).

Кодирование — процесс преобразования знаков и символов в числа, по историческим причинам эти числа различаются в зависимости от системы кодирования. Например, ASCII относится к тем временам, когда в англоязычных странах все думали: «Ну, 7 бит, пожалуй, будет достаточно». Поэтому при наборе текста на отдельных языках (немецком, польском и тем более китайском) некоторые символы могут не отображаться. Позже было придумано множество дополнительных систем кодирования, таких как ISO-8859-1 (для некоторых восточноевропейских языков) или Windows-1252.

Однако проблема серьезнее, чем кажется на первый взгляд. Наш мир все время меняется, и приходится подстраиваться под новые условия, к тому же кодировка знаков отнюдь не самое легкое занятие.

На сегодняшний день унифицирование всех знаковых систем важнее экономии памяти. Поэтому последним этапом развития систем кодирования можно назвать Юникод. При возможности хотя бы пробежитесь по статье в «Википедии», посвященной Юникоду, чтобы понять, каких размаха и сложности эта система.

Во многих языках программирования названия функций и переменных в коде должны начинаться только с букв или некоторых символов, таких как «_», но не с чисел или других знаков. Юникод, так же как и ASCII, различает буквы, цифры и знаки, но если буквы из разных знаковых систем, это может затруднить распознание. Во многих языках программирования вы можете назвать переменную Δ, так как это буква греческого алфавита, но не можете дать переменной название ☺, так как это уже символ. Однако все равно лучше использовать латинские буквы, чтобы не сбивать с толку программиста, читающего код. Юникод лучше применять для написания текста, а не для кодов.

Не следует использовать систему ASCII. При применении библиотек программирования иногда можно выбирать кодировку, которую будет использовать функция. Вы не ошибетесь, если выберете UTF-8, даже если поклялись себе всегда использовать ASCII. Со временем обязательно появится какой-нибудь человек с логином Świętobor Jabłoński. Тогда вы и поблагодарите себя за когда-то правильно сделанный выбор.

В качестве утешения: кодирование знаков — одна из немногих областей программирования, в которой программисты не из англоговорящих стран оказывают-

ся в более выигрышной ситуации. Они относятся к этим вопросам еще более небрежно. Стоит заметить, что проблема постепенно исчезает, так как появляется все больше программ с использованием Юникода по умолчанию.

Указание времени и даты

Область применения: в основном Интернет.

Последствия неведения: индийских пользователей вашего сайта будет выбрасывать сразу же после входа, так как сеансы будут заканчиваться из-за недействительности.

Краткая информация: хоть время и линейно, однако ежедневно используемое нами указание времени таковым не является. Из-за наличия так называемых секунд координации и переходов на зимнее и летнее время оно способно делать скачки в будущее или даже прошлое. Таким образом, вы не можете быть уверены в том, что каждая минута составляет 60 секунд, иногда их может быть и 61. В месяце не всегда бывает 30 дней, а в году — 365.

В одном дне не всегда одинаковое количество часов. Например, при переходе с летнего времени на зимнее час в промежутке между 2:00 и 3:00 проходит дважды — по летнему и, после перехода, по зимнему времени. Первый час (с 2:00 до 3:00 по среднеевропейскому летнему времени) официально обозначается 2A, а второй час (с 2:00 до 3:00 по среднеевропейскому времени) — 2B.

Время в компьютере еще более неточно. Большинство компьютеров имеет встроенные кварцевые часы, которые периодически сверяют время с сервером единого времени. Встроенные часы довольно неточные, поэтому заметно отстают от единого времени до того момента, пока не соединятся с сервером. Если вы используете системное время для синхронизации или для точной работы вашей программы, уточните, использует ли программа линейное время, то есть то, которое не надо сверять и сбрасывать.

Некоторые процессоры, в особенности в смартфонах, не имеют предварительно настроенного соединения с сервером единого времени, из-за чего время на смартфоне может не совпадать на несколько секунд или даже минут с установленным временем. Если вы пишете веб-приложения для back-end-сервера, то будьте готовы к тому, что время на смартфоне и время сервера могут не совпадать. Если вам когда-нибудь придется писать мобильные приложения, помните о том, что часовой пояс может в любой момент поменяться, например, если пользователь зашел в приложение из Пекина.

Хоть сейчас и существует стандарт регулирования времени во всем мире — Всемирное координированное время (UTC), однако, работая с временем и датой, следует в первую очередь обращать внимание на часовые пояса.

Если ваше приложение фиксирует время пользовательской активности или других операций, то его можно преобразовать в UTC. Это легко сделать, если ваша программа работает исключительно с UTC-форматом и если внешние временные показатели конвертируются и сохраняются в UTC. При отображении времени обязательно указывайте часовой пояс.

Timestamp (дословно — отметка времени) — момент, когда произошло определенное событие. Самая часто используемая система отметки времени — Unix-Timestamp. Она ведет отсчет с 01.01.1970. Создатели назвали этот день, который приблизительно совпадает с возникновением системы, началом новой эпохи. Между тем в каждой операционной системе существуют также системные часы, которые являются более точными и высчитывают время до миллисекунд с начала 1970 года. Обыкновенная секундная временная отметка представляет собой десятичное число, миллисекундная отметка — тринадцатизначное. Если вы получаете грубые ошибки в указании календарного числа и времени, то, возможно, неправильно конвертировали значения секундной или миллисекундной временной отметки в дату. Или же вы конвертируете временные отметки в UTC по местному времени, из-за чего время и число могут отображаться неправильно.

Существует довольно много форматов записи календарных дат. Всегда сохраняйте числа и время в одном формате и по возможности используйте кодировку ISO-8601 (год, месяц, день в формате ГГГГ-ММ-ДД и время с указанием часового пояса). В отличие от системы Unix-Timestamp этот формат будет понятен каждому.

В качестве утешения: сейчас в каждой области программирования существуют различные программы для конвертации местного времени в формат UTC и наоборот. Если повезет, найдете и такую, которая сможет конвертировать числа из григорианского календаря в юлианский и обратно.

Сохранение десятичных дробей в виде строковых, целочисленных и десятичных переменных

Область применения: используется редко, однако везде.

Последствия неведения: используя некорректные числа, в худшем случае можно нажить проблемы с финансовым отделом.

Краткая информация: компьютер сохраняет числа в бинарной системе, люди считают в десятичной. Так как число 10 не подпадает под двоичный порядок, то оно не может быть сохранено в бинарной системе, и таких чисел много. Так, например, число 0,1 сохраняется в большинстве языков программирования как 0,09999... а при отображении снова округляется до 0,1.

Даже если бы компьютеры работали в десятичной системе, они все равно не сохраняли бы все числа в первоначальном виде. Переменные целого типа имеют ограниченный диапазон значений и не могут иметь цифру после запятой, однако их можно отобразить точно, не прибегая к округлению. Что же касается чисел с плавающей запятой, то тут диапазон значений гораздо шире за счет наличия цифр после запятой, в данном случае без округления отобразить не получится. Если использовать переменные с плавающей запятой при вычислениях, то неточности будут суммироваться. В обоих случаях выбор типа переменной зависит от свободной оперативной памяти, которой вы располагаете.

Поскольку мы редко оперируем десятичными дробями с большим количеством знаков после запятой, округление не кажется нам большой проблемой до появления суммирующихся погрешностей. Удивительно, но неточная передача чисел может создать проблему при использовании стандартного диапазона чисел.

Пример, который наглядно демонстрирует проблемы, возникающие при преобразовании десятичных чисел в бинарные:

```
double a = 0,7;
double b = 0,9;
double x = a + 0,1;
double y = b – 0,1;
if (x == y) {
    /* Это сравнение не сработает в большинстве популярных языков */
}
```

Существуют различные библиотеки, которые могут обрабатывать десятичные дроби с самым разным диапазоном значений. В некоторых языках существуют такие типы данных, как `Decimal` и `bignum`, специально для чисел таких типов. Если вы работаете с денежными суммами или вам нужна предельная точность значений по другим причинам, рекомендуем ознакомиться с этими библиотеками.

Если же у вас есть возможность полностью переключиться на целые числа, то таким образом вы можете избежать описанных проблем. Например, при ведении обычной бухгалтерии без учета процентов валютные суммы, как правило, долларов или евро ограничены двумя знаками после запятой. А это значит, что вы можете сохранять десятичные дроби как целые числа, просто умножая на 100 и отображая сумму в центах, а не в евро.

Также будьте осторожны, экспортируя данные или импортируя из других источников, так как в некоторых странах используются разные знаки для разделения дробной и целой частей числа. Например, в Европе применяют запятую, а в США — точку. В программировании используется американская система, в бухгалтерии — в основном европейская. Программы для работы с электронными таблицами при экспорте в CSV (см. главу 24) также используют европейский формат.

Тот, кому довелось поработать в транснациональной корпорации, вероятно, не раз чувствовал на себе эту разницу, пытаясь импортировать данные другого формата.

С возникновением Всемирной паутины заметно возросла роль текстовых форматов (XML, JSON, SOAP), в которых эта проблема проявляется еще более явно, чем в ранее используемых бинарных форматах. По этой причине, если вы обмениваетесь данными между различными системами или числовыми форматами, отказ от использования чисел с плавающей запятой был бы вариантом решения проблемы. В таком случае вы можете переносить целое число и отдельно знаменатель. Если это не выход, то используйте американскую систему и будьте готовы к записи чисел в экспоненциальной системе в более сложном формате (2e-4 вместо 0,0002). К счастью, с числами такого типа без труда справится функция `parseFloat()`, которая существует во всех языках программирования.

В качестве утешения: разработчики языков сами частенько путаются в этой области. В 2011 году в Java и PHP всплыла ошибка в сохранении десятичных

дробей, из-за которой произошли сбои во многих веб-приложениях. Больше об этом можно прочесть в статье www.theregister.co.uk/2011/01/04/weird_php_dos_vuln/ (на английском языке).

Передача переменных в виде значений или ссылки

Область применения: везде.

Последствия неведения: путаница при работе с программой.

Краткая информация: переменная и величина не являются одним и тем же. Повторяйте это себе как можно чаще и пытайтесь осмыслить. Предложение звучит таинственно поначалу и фундаментально потом.

Например:

```
var1 = 5;
var2 = var1;
```

Интуитивно все могут предположить, что `var2` равняется 5, так как `var1 = 5`. Однако это не так:

```
function confuse (local_var) {
    local_var = 7;
}

var1 = 5;
confuse (var1);
```

Здесь задача усложняется. Какой величины теперь `var1`, 5 или 7?

В большинстве языков `var1` будет иметь значение 5, так как переменная `local_var` является локальной переменной (см. главу 26). Значение `var1`, но не сама переменная `var1` при вызове функции копируется в `local_var`. Таким образом, наделение `local_var` значением 7 никак не отразится на значении `var1`:

```
function confuse ($local_var) {
    local_var->firstName = "Bernadette";
}
```

Теперь функция вызывается следующим образом:

```
var1 = new User {
    firstName => "Bernd",
    lastName => "Lauert"
}
confuse (var1);
```

Какое же имя получила переменная `var1`? Интересно, что в большинстве языков переменную называют `Bernadette`. В отличие от простых переменных, например целых, при вызове которых функции передается лишь само значение составных переменных, таких как структуры, массивы, объекты, а также частично строковые переменные, в ней содержится ссылка на локальную переменную. Это объясняет-

ся тем, что var1 является не самим массивом, а лишь указателем на него. Таким образом локальная переменная local_var как бы внедряется в переменную var1 и становится одним целым. Поэтому при изменении local_var изменяется и var1.

Причина этих различий весьма прагматична — если имеется массив более чем с миллионом данных и в нем дается ссылка на функцию, то в функцию передается лишь пара байтов, которые занимает адрес ссылки на содержимое массива. Если бы массив передавался через значение, то все данные приходилось бы перемещать, а при большом количестве данных это было бы неописуемо медленно. Простые же типы данных часто передаются в качестве обычных значений, так как они не занимают много места. Было бы не очень удобно передавать их в виде ссылок.

Говоря в общем, различие между простыми и сложными типами данных является целесообразным также потому, что, говоря о числах, мы не думаем о неизменяемых константах. Число 5 для нас неизменно. Поэтому, если я присвою переменной со значением 5 значение 3, то я захочу поменять не саму сущность 5, а просто значение переменной. Если участок данных находится в массиве, нужно всегда иметь возможность изменять некоторые данные (например, имя или фамилию).

Существуют языки с четким разграничением типов данных, которые можно передавать с помощью ссылки и с помощью значений. А существуют и более гибкие в этом отношении языки, например, в C++ переменной присваивается ссылка, которая передается в функцию:

```
void square(int x, int& result) {
    result = x * x;
}
```

Эту функцию можно вызвать следующим образом:

```
int result = 0;
square (5, result);
```

Даже если у функции нет выводных значений, у result при запуске в любом случае значение 25, поскольку & указывает не присваивать функции значение переменной result, а просто ссылаться на эту переменную.

VOID

Ключевое слово void указывает на то, что переменной не может быть присвоено значение result = square (5).

Существуют языки, в которых четко обозначено, какие переменные можно вписывать в функции, помечая их, к примеру, как IN OUT. Таким языком является язык баз данных СУБД, который используется как встроенный язык для Oracle Forms и поэтому играет важную роль в больших корпорациях. Будьте осторожны при изменении ссылок или соответственно IN OUT-переменных в функциях. Вам будет легче разобрать свой код, если вы будете писать функции по шаблону «ввод — функция — вывод» (в данном случае var x = function(y)).

Разные языки программирования сохраняют ссылки и значения по-разному. Изучая новый язык, следует выделить время, чтобы разобраться с этим. К сожалению, во многих языках трудно понять вопросы дифференциации ссылок и значений.

Например, Ruby является современных языком, который все передает в виде ссылок, однако часто это выглядит как передача значений. Если же начать интересоваться этим вопросом, то можно наткнуться на обширные дискуссии, в которых профессиональные программисты не могут прийти к единому выводу, в каком виде язык выдает переменные. Java же, наоборот, всегда выдает значения, однако многие программисты готовы поклясться, что выдаются ссылки на объект.

Резюме: если вы не осилили этот пункт, то запомните всего одну вещь — переменные, которые передаются с помощью ссылки в функцию, могут ими же и изменяться. Если же вы передаете в функцию само значение, то переменная постоянна (immutable).

В качестве утешения: если не понять принципа работы, то методом проб и ошибок вы все же сможете работать с программами средней сложности. В Сети полно комментариев, из которых ясно, что даже продвинутые программисты не понимают принципа обработки переменных в том или ином языке.

Менее утешительным является то, что непонимание принципа работы зачастую вызывает ошибки, которые впоследствии трудно решить.

Тяжелая работа с ничем

Область применения: базы данных, также применяется во многих языках.

Последствия неведения: лишняя морока.

Краткая информация: `NULL` является не тем же самым, что цифра 0, а чем-то вроде вакуума, что означает, что там, где он стоит, не может быть вообще никакого значения, даже 0. Грубо говоря, `NULL` соответствует не 0, а «неизвестному». К сожалению, в разных языках значение различается, а в некоторых языках (например, СУБД) оно еще зависит от контекста, в котором этот самый `NULL` употребляется. Идея «неизвестного» в разных языках выражается по-разному. К примеру, `null` в Java, `nil` в Ruby и `None` в Python.

Во многих языках переменным без заданного значения присваивается значение `NULL`. В некоторых языках (JavaScript) переменную такого типа называют `undefined`, в Perl — `undef`. Однако это не значит, что в этих языках не может быть `NULL`, он просто будет передан по-другому.

В некоторых языках существует похожее выражение — `NaN` (Not a Number) или `Infinity`, которые обозначают нерациональное значение, то есть что-то, что не выражено цифрой, а значит, что-то бесконечное. `NULL`, `undefined` и `NaN` значительно отличаются друг от друга, однако общее у них то, что их нельзя использовать в математических операциях. Зачастую появляется сообщение об ошибке или же снова `NULL`, `undefined`, `Infinity` или `NaN`.

Но будьте внимательны. Например, в JavaScript существуют математические различия между `undefined` и `null`:

```
>1 + undefined
NaN

>1 + null
1
```

В данном контексте NULL будет представлять собой значение 0.

В СУБД при сравнении нельзя использовать NULL в качестве числового значения (например, select *from users where firstname = NULL). Надо выбрать другой оператор сравнения: select * from users where firstname is NULL. Это чем-то похоже на язык Perl — здесь надо прописывать if defined ($var), чтобы проверить, является ли $var undef. В Java и многих других языках можно, наоборот, без проблем тестировать с заданным значением NULL: if myObject ==null {....

Тот, кто при запросе базы данных пишет select * from users where firstname = NULL, будет удивлен, так как запросы СУБД, которые используют = и NULL, не выдают результат. Странно, но запросы, содержащие <, > и NULL, также не выдают результат, так как любое сравнение с NULL дает NULL.

В объектно-ориентированных языках программирования NULL может называться объект, для которого еще не был вызван конструктор. Если же попытаться получить доступ к значениям или же методам такого объекта (например, print user.firstname), то программа будет завершена сообщениями об ошибке NullpointerException, так как NULL не имеет firstname.

В переменных типа Boolean, которые могут содержать в себе лишь true или false, может не стоять значение, в таком случае оно будет NULL. Часто это надо учитывать при написании условий с операторами if/then/else. Так, например, в СУБД сравнение с NULL всегда выдает NULL, в Java — false. В JavaScript сравнение типа if myVar == null всегда выдает false, однако если у переменной значение undefined или null, то операция проходит успешно. В JavaScript и PHP существует также оператор ==, который при таких сравнениях выдает false при значении переменной undefined и true при значении null.

В качестве утешения: значение NULL в переменных часто приводит к явным и хорошо заметным ошибкам.

Менее утешительным является то, что точное значение NULL зависит от языка. Поэтому при изучении нового языка будет полезным детальное ознакомление с этой темой.

Рекурсия

Область применения: обработка иерархических данных (например, подкатегории), в общем, все древовидные структуры.

Последствия неведения: программа расходует все доступное пространство в стеке и таким образом увеличивает нагрузку на процессор до 100 %, из-за чего компьютер работает слишком медленно. Это происходит из-за того, что раз за разом вызывающая себя функция написана с неправильным условием отмены. Она может вызываться бесконечно долго до тех пор, пока не закончится память.

Краткая информация: с помощью рекурсии можно написать компактный и красивый код, если речь идет о древовидных иерархических структурах.

В принципе, можно обойтись и без рекурсии, однако это будет занимать гораздо больше времени.

Решая проблему с помощью рекурсии, следует обращать внимание на максимальный объем обрабатываемых данных. В то время как простые функции при

растущем количестве данных начинают работать медленнее, рекурсивные в таких случаях в конце концов приводят к сбою в программе.

Следует определить предельно допустимый объем данных для рекурсии и стараться придерживаться его.

Далее приведен пример рекурсивной функции, которая бесконечно выдает строковые данные:

```
main () {
    print "ich bin eine Funktion, ";
    subfunction ();
}
function subfunction () {
    print "die eine Funktion aufruft, ";
    subfunction ();
}
```

Из примера заметно, что с парой строчек кода можно создать бесконечно много данных. Или же в худшем случае просто сломать процессор, а в лучшем — привести к сбою программы.

В качестве утешения: неправильно заданная рекурсия бросается в глаза при первом же тестировании. В отличие от некоторых других ошибок, при неправильно заданной рекурсии часто высвечивается ошибка переполнения стека.

Юзабилити

Область применения: везде.

Последствия неведения: пользователи вашей программы проводят слишком много времени, гадая и пытаясь разобраться, как работать в программе. Вы как разработчик проводите больше времени, чем нужно, занимаясь техподдержкой и исправлением дефектов, вызванных пользователями. В конце концов вы разочаровываетесь в умственном развитии всего человечества и становитесь активным пропагандистом консервативных идей.

Краткая информация: если вы работаете с приложениями или веб-сайтами, которыми пользуется большое количество людей, есть смысл ознакомиться с литературой или блогами, посвященными юзабилити. Мы приводим вкратце самое важное.

Пользователь всегда прав. Если при обращении с вашей программой он ведет себя как дикарь, вы, конечно, имеете полное право возмущаться, но только наедине с собой, сидя в комнате. И вы не можете написать гневный пост в блоге, даже не можете обсудить это в узком кругу сотрудников. Так как это ваша программа, ее непрактичность и непродуманность вынуждают пользователей неправильно работать с приложением. Если люди хотят улучшить свои умственные способности, они разгадывают судоку или записываются на онлайн-курсы в Массачусетский технологический. Приложение же не должно заставлять людей выполнять тяжелые мыслительные операции, а должно быть настолько простым, чтобы даже дикари из каменного века могли в нем работать.

Пользователь никогда *не* виноват. Повторяйте себе это как можно чаще.

Если по определенным причинам существуют устоявшиеся способы создания сайта (например, движок прокрутки, формулировка «Вход/Регистрация», условность, при которой после щелчка на шапке сайта попадаешь на главную страницу), то, пожалуйста, не пытайтесь быть оригинальным. В очень редких случаях такие инновации могут быть оправданны. Если вы не главный разработчик компании Apple, это не тот случай.

Старайтесь быть объективным и не судить всех по себе. Возможно, пользователь вашего приложения — дальтоник из страны, где пишут справа налево, и разрешение экрана его кофеварки (японские модели с доступом к Интернету) — 30 × 1280. Наверное, пример немного утрирован, но всегда думайте о том, что у пользователей не такое же устройство, на котором вы тестируете программу, и они находятся в других условиях.

Важное и общепринятое всегда должно быть четко отображено на сайте. Специфическое и необычное располагается в менее заметном месте.

Дополнительных объяснений, как правило, не требуется. В противном случае их размещают сразу под объясняемым объектом. Сообщения о некорректно заполненном формуляре должны отображаться никак не вверху или внизу страницы, а непосредственно напротив поля в анкете. Подсказки не должны содержать неактивные ссылки на документы, понятия, объекты, чтобы пользователям не приходилось самим искать их по всему Интернету.

В качестве утешения: несмотря на прогресс последних лет, программисты все так же мало времени уделяют вопросам юзабилити. Так что ваши мелкие погрешности могут никому и не броситься в глаза.

18 Компромиссы

> С какого значения должен начинаться массив — 0 или 1? Мой компромиссный вариант 0,5 отвергли, на мой взгляд, это было недальновидно.
>
> *Стэн Келли-Буттл, британский музыкант и программист*

В пособиях по программированию мир выглядит так, словно существует всего один истинный путь — достаточно всего лишь ему следовать. Но когда беседуешь с программистами или читаешь их блоги, большинство признается, что им приходится применять собственные своеобразные методы. Однозначные решения, которые устраивали бы всех в долгосрочной перспективе, — скорее исключение, чем правило.

Дело в том, что слово «решение» — лишь красивый эвфемизм, означающий «откладывание проблем». В одном месте можно избавиться от чего-нибудь неудобного или недопустимого, но обычно это означает, что осложнения возникнут в другом месте. Джон Галл, написавший исключительно полезную книгу *Systemantics*[1], помогающую понимать сложные системы, говорит об этом так.

> Мнение о том, что ошибки исчезают, поскольку детали становятся все безотказнее, — это, естественно, чистейшее выдавание желаемого за действительное. Лишь банальнейшие ошибки могут быть связаны с (не)надежностью деталей. Если ошибка возникает один раз на миллион случаев, то это не проблема для компьютеров, производители которых банкротятся в каждом третьем случае. В результате покупателю приходится иметь дело с полноценной системой, техподдержка которой уже не осуществляется. В целом повышенная надежность деталей лишь перераспределяет анергию[2] на связующие узлы системы или другие детали, починить которые уже сложнее. Существует правило «если что-то не сломалось, значит, сломалось что-то в другом месте».
>
> *Джен Галл*

---

[1] *Gall J.* Systemantics. — How Systems Work and Especially How They Fail.

[2] Анергия — в систематической теории Галла, противоположность энергии. Анергия накапливается в тупиковых ситуациях, подобно тому как в сжатой пружине сосредотачивается энергия.

Можно переформулировать проблему так, что с ее новой версией будет гораздо легче управиться. Если немного повезет, проблему можно видоизменить так, что в ней будет проще разобраться и многим другим людям. Но и здесь действует правило «Бесплатный сыр бывает только в мышеловке». В каком-то отношении новая версия окажется хуже старой. Если же вашим кодом пользуется кто-либо кроме вас, то для него это слабое место может оказаться настолько важным, что он примет реорганизацию за ошибку. Чаще всего приходится обдумывать следующие моменты.

- Классический треугольник управления проектами: проект должен выполняться быстро, дешево и результативно. На практике приходится удовлетвориться максимум двумя этими качествами.
- При разработке ПО нередко встает вопрос о том, будет ли проект вообще доведен до конца. То есть приходится выбирать между посредственным кодом, который, однако, действительно доживет до релиза, и надежным кодом, который никогда не будет завершен.
- Программное и аппаратное обеспечение с множеством конфигурационных возможностей открывает огромные возможности перед немногочисленными технарями-энтузиастами. Если ограничить набор доступных функций, то приложение становится удобным для более широкой пользовательской аудитории, однако работать с программой они могут лишь в ограниченном объеме.
- Небольшие аккуратные системы способны на малое. Чтобы достичь результата, из них сначала нужно возвести большую угловатую структуру.
- В большинстве случаев удается либо сократить количество применяемых методов, классов, файлов, либо сделать их компактнее, но не решить обе эти задачи сразу.
- Хорошие комментарии делают код понятнее. Плохие, неверные, устаревшие, ошибочные комментарии только ухудшают код. Со временем хорошие комментарии превращаются только в плохие.
- В большинстве языков локальные переменные можно объявлять либо в начале функции, либо в том месте, где они могут понадобиться, то есть прямо в теле функции. У обоих способов есть свои достоинства и недостатки. В первом случае мы аккуратно перечисляем все переменные в одном месте, но это место оказывается далеко от того участка кода, где эти переменные будут использоваться. Во втором случае код получается более путаным, но только переменная объявлена — и вот она уже используется.
- Мягкое кодирование (http://thedailywtf.com/articles/Soft_Coding): в принципе, хорошо, если мы не зашиваем в программу при компиляции такие вещи, как управляющая запись, а храним их в файлах настроек, поскольку подобные элементы могут меняться. Однако следует основательно обдумать, что произойдет, если пользователь (или администратор) изменит значение. В таком случае все вычисления сразу будут выполняться с учетом новой управляющей записи (большая катастрофа), либо админ должен вносить новую управляющую запись ровно в 00:00 31.12, чтобы с этого момента все вычисления уже велись с новой записью, либо существует возможность составить хронологический список управляющих записей. Иногда проще все жестко закодировать, а когда детали изменятся, написать новую версию программы.

- Плохому программисту, отыскивающему в коде определенное место, приходится всякий раз просматривать весь код заново. Если же последовать простому совету и разбить код на множество файлов, то в некоторых случаях потребуется искать еще дольше.

Если определенная практика или соглашение долгое время оспариваются, это явный признак того, что мы столкнулись как раз с таким неоднозначным моментом. Просто не удается прийти к общему мнению, стоит ли выкурить проблему из одной части системы и переместить в другую.

Поскольку все люди имеют собственные мнения и предпочтения, они пишут разный код, и это даже хорошо. Подобно генетическому разнообразию в биологии, разнообразие кода в программировании делает всю отрасль более зрелой и обеспечивает ее дальнейшее развитие.

Обманчивые достоинства

Когда у плохого программиста выдается хороший день, в который хочется все исправить и оптимизировать код, случается так, что для достижения верных целей применяются неверные средства. Вспомните, как хорошо звучат предвыборные речи политиков, но либо результат получается нулевым, либо возникают новые проблемы, а старые при этом не решаются. Здесь мы рассмотрим три таких обманчивых достоинства, как уверенность в завтрашнем дне, оптимизация скорости и красота.

Уверенность в завтрашнем дне

Вы написали код, которым очень гордитесь, поскольку он не только отвечает всем текущим требованиям, но и учитывает возможные доработки завтрашнего и даже послезавтрашнего дня.

Однако знакомые опытные программисты, увидев ваш код, восклицают: «YAGNI!» В переводе с английского эта аббревиатура означает: «Вам это не понадобится». Ваша программа сложнее, чем требуется, ради этого потребовалось выполнить лишнюю работу, в программе увеличилось число багов, а исправлять ее сложнее. Прежде всего возникает опасность, что ваши инвестиции никогда не окупятся, ведь гадать о будущем — дело неблагодарное. Тогда вы будете страдать от неудобств собственного творения, так и не воспользовавшись его достоинствами.

Поэтому пишите лишь тот код, который действительно требуется для решения задачи. Если когда-нибудь в самом деле возникнет ситуация, которую вы сейчас предвосхищаете, то вы уже будете гораздо умнее и сможете решить задачу эффективнее, чем представляете себе это сейчас. Кроме того, возможно, к тому времени вы будете располагать совершенно иными техническими возможностями. Наилучшая подготовка к будущим требованиям — писать понятный удобочитаемый код. А пока это будущее не наступило, отбросьте все, что в данный момент не нужно. Жизнь станет от этого проще и прекраснее.

Важнейшие признаки YAGNI таковы.

- **Абстракции и обобщения.** Наряду с YAGNI существует принцип «Вы больше никогда этим не воспользуетесь». Неопытные программисты часто оказываются в такой ситуации: когда потребуется заново использовать имеющийся код, уже будет найдено значительно более удачное решение. Даже бывалый программист частенько обнаруживает, что весь мир давно знает более рациональное решение, чем получилось у него.
- **Огромное множество конфигурационных возможностей.** Часто, особенно при написании программ с графическим пользовательским интерфейсом, наступает такой момент, к которому в программу уже встроены все мыслимые конфигурационные возможности, либо потому что это считается необходимым, либо если этого требуют пользователи. Конфигурационное меню, которое на данном этапе становится относительно стабильным, оказывается столь запутанным, что ни один пользователь уже не в состоянии что-либо сконфигурировать. Вместо этого нужны разумные умолчания, а круг доступных функций должен быть ограничен самыми необходимыми вариантами. В теории все это, к сожалению, звучит гораздо проще, чем оказывается на практике, поскольку, чтобы задать хорошие стандартные настройки, нужно вплотную познакомиться с привычками и потребностями пользователя.

Полезно сразу отбросить версию, что какие-либо правильные решения могут приниматься без подготовки. Даже специалистам не всегда под силу правильно определить, какой объект должен за что отвечать или какое имя должно быть у функции. Итак, можно без зазрения совести для начала что-нибудь смастерить, а потом довести это до ума либо расставить все по своим местам. При этом будут полезны приемы и инструменты, минимизирующие издержки на последующие изменения. Но это ни в коем случае не оправдывает порочных практик. Если вы хотите называть свои переменные `dings`, то ссылаться на нас не вправе (см. главу 5). Мы лишь хотим сказать, что если на протяжении карьеры вы примете пару неидеальных решений, которые потом можно будет откорректировать, то ничуть не покроете себя позором.

Скорость

Если вы помните, что такое матричный принтер, то, вероятно, припоминаете и так называемый быстрый режим, или режим черновика. Поскольку такая технология обеспечивала либо хорошую удобочитаемость, либо высокую скорость работы, но не то и другое сразу, пользователю приходилось выбирать. Либо принтер пыхтит и печатает удручающе медленно, но задействует все свои возможности — и получается меню на мраморной бумаге для дорогого ресторана, либо он строчит на бумажном рулоне выписки из счета за последние 20 лет, но эту скоропись едва можно прочитать.

Когда пишешь код, скорость и удобочитаемость тоже редко достигаются одновременно. Оптимизированный код — это уже не простейшее решение поставленной задачи, его сложнее прослеживать, что затрудняет поиск ошибок. При оптимизации

скорости можно легко заиметь лишние проблемы, так и не воспользовавшись достоинствами такой быстрой работы.

Поэтому всякий раз, когда при программировании вам приходит на ум идея о производительности, гоните ее прочь. Вероятность того, что внутренний голос вас обманет, исключительно высока хотя бы потому, что 80 % кода в вашей программе будет выполняться столь редко, что можно абсолютно не учитывать, сколько времени на это уходит, 2 или 20 мс. К тому же даже опытным программистам едва ли удается угадать те 20 % кода, от которых в самом деле будет что-то зависеть.

Если довериться интуиции, то всегда будешь оптимизировать ту часть кода, которую оптимизировать удобнее всего (как только станешь программистом поопытнее). При оптимизации кода на скорость действуют те же правила, что и при походе на стройрынок: всегда нужно что-то заранее измерять (см. раздел «Тестирование производительности» главы 16), особенно те вещи, которые, как кажется, измерять-то и незачем.

Будучи начинающим программистом, можете смело отложить подобные планы на потом. Если вы будете тратить время и ресурсы на оптимизацию, если не вполне понимаете устройство компьютера или компилятора, то это верный путь к провалу. Если же вы, несмотря ни на что, хотите продолжить самообразование, то вам, вероятно, помогут следующие железные правила.

- Код должен быть тщательно протестирован, чтобы в нем не осталось известных проблем. Оптимизация и исправление ошибок плохо сочетаются друг с другом.
- Оптимизация кода, как правило, приводит к новым ошибкам, которые потом тоже придется править, — сразу выделяйте на это дополнительное время.

Познакомьтесь с библиотеками для тестирования производительности (см. главу 16) и выясните, что пишут в Интернете о возможностях оптимизации при работе на интересующем вас языке. Есть такие языки, в частности JavaScript, оптимизация которых — известная головная боль. Дело в том, что среда исполнения таких языков (в случае с JavaScript это браузер) претерпевает бурную эволюцию. Сегодняшняя оптимизация уже завтра может оказаться палкой в колесе.

Если вы программируете на компилируемом языке, например на C или C++, то в самом начале работы можете указать компилятору, чтобы при трансляции кода он задействовал более высокую степень оптимизации. В средах разработки в настройках проекта для этого предусмотрены специальные флажки, в случае с командной строкой они называются флагами. После того как код будет транслирован с высокой степенью оптимизации, его обязательно нужно протестировать повторно, поскольку на некоторых шагах оптимизации могут вкрасться ошибки. Однако не переоценивайте тот выигрыш в скорости, которого можно достичь путем агрессивной оптимизации, — как правило, он составляет всего пару процентов.

Для многих языков существуют так называемые инструменты-профилировщики. Они анализируют код статически или динамически и при оценке показывают те его участки, на выполнение которых программа тратит слишком много времени либо к которым регулярно возвращается.

Следует оптимизировать только те участки, которые профилировщик оценивает как горячие точки (hotspot). Отдаленные участки кода, которые выполняются редко, даже при мастерской оптимизации не дают заметного ускорения работы программы.

Если нет желания разбираться с профилировщиком, но у вас есть отладчик или иная возможность остановить работающую программу и посмотреть, на каком месте она оказалась, то можно задействовать простой прием, который в командной строке выполняется командой `kill`, а в среде разработки — с помощью встроенного отладчика. В расчете, что нам повезет, мы пару раз приостанавливаем выполнение программы и в каждом из этих случаев просматриваем стек вызовов. Если существует проблема с производительностью, на которую тратится, допустим, 80 % времени исполнения, то примерно в 80 % случаев, приостановив программу, мы очутимся именно в этой точке.

Программы с графическим пользовательским интерфейсом можно существенно ускорить, если направить все усилия по оптимизации именно на компоненты интерфейса. При этом программа быстрее реагирует на действия пользователя и субъективно кажется, что она ускорилась. Фоновые вычисления, при которых индикатор загрузки растет на 10 % медленнее, воспринимаются гораздо терпимее.

В случае с веб-приложениями обычно очень важно, какое время тратится на передачу данных из браузера на сервер и обратно. В данном случае оптимизация обычно сводится не к настройке производительности сервера, а к тому, чтобы уменьшить объем передаваемых данных. Для этого на веб-сервере может включаться deflate-сжатие, могут использоваться уменьшенные файлы JavaScript или CSS либо сокращаться количество картинок или уменьшаться их размер. Следующий наиболее действенный шаг оптимизации — упростить структуру HTML, чтобы веб-страницы отображались быстрее (сравните с вышеупомянутым пунктом).

Если инструмент не имеет графического интерфейса и не подключается к Интернету, бывает достаточно разбить задание на пакеты таким образом, чтобы их можно было параллельно обрабатывать на многоядерном процессоре. Но для этого, как правило, нужно разбираться в многопоточности. А многопоточность — явно не то дело, которым следовало бы заниматься начинающему программисту.

Совершенство, красота, элегантность

Красота кода во многом подобна телесной красоте: если немного повезет, то внешне вы можете выглядеть довольно привлекательно, но под кожей все равно кровавое мясо, которое показывают в фильмах ужасов. Чтобы добиться некоторой красоты кода (как и человеческого тела), не требуется ничего сверхъестественного, но стремление к совершенству — это занятие «на фултайм». Причем даже в самом красивом коде через несколько лет никто уже не разберется.

Конечно, приятно было бы решить проблему раз и навсегда, прежде чем переходить к выполнению новой задачи. Но к концу проекта отдача от дальнейших оптимизаций постепенно стремится к нулю. Плохой программист слишком рано бросает такие доработки и считает достаточно хорошим решение, которое на деле является в лучшем случае половинчатым. Но если запоздало прекратить оптимизацию или не прекратить ее вообще, то можно поставить под угрозу весь проект. Нужно уметь рано или поздно сделать последний штрих и перейти к разработке другого фрагмента, на который действительно целесообразно тратить время.

Программирование не конкурс красоты. Это постоянная необходимость иметь дело со сложностями, разбираться в путанице неясных, противоречивых и изменяющихся требований, а также преодолевать ограниченность собственных возможностей. И так до смерти. Как и в жизни, в этой работе требуется признавать несовершенство и трудиться в таких условиях.

Именно стремление к совершенству то и дело подбрасывает новые фатальные идеи, о чем мы уже говорили в главе 15: «Просто устроить генеральную уборку! Начать все с начала! На этот раз все сделать правильно!» — когда напрашивается именно такая идея, привлекающая вас своей красотой, элегантностью и безупречностью, не поддавайтесь. В тяжелых случаях вам, пожалуй, может понадобиться своеобразный спонсор в том значении, которое принято в среде анонимных алкоголиков. Просто обратитесь к другу и скажите: «Помоги мне! Меня одолевает навязчивая идея стереть весь код и начать все сначала». Хороший друг вас от этого удержит.

Код никогда не совершенен. Однако это же означает, что он никогда не безнадежен. Сегодня ваш код неказист, возможно, он будет таков и через десять лет. Но если вы отныне и навсегда решите, что создание красивого, элегантного кода — ваша первоочередная жизненная цель, то, возможно, в далеком будущем вам удастся создать пару строк, о которых и через несколько лет можно будет сказать: «Неплохо». Но мир явно не станет хуже, если вместо этого вы признаете, что на свете нет ничего совершенного, и удовлетворитесь написанием такого кода, который будет вполне понятен и через шесть месяцев.

Абсолютизация: когда допустимы порочные практики

Во всех книгах, пособиях и подкастах на тему программирования постоянно подчеркивается, как важно писать чистый код, пользоваться системами контроля версий и в принципе поддерживать порядок. В принципе — это хорошо и правильно, и следующий раздел ни в коем случае не должен подтолкнуть вас к неаккуратной работе и отбить интерес к самосовершенствованию.

Но всегда есть исключения.

Сознательно писать плохой код — все равно что делать долги. Это порицается, но в определенных жизненных ситуациях у вас просто нет другого выхода. И как кредиты впоследствии приходится отдавать с процентами, вы расплачиваетесь за технический долг в коде, допущенный ради скорейшего достижения цели. Вам предстоит неблагодарное занятие — расплетать безумную путаницу в коде, выносить настройки в файлы и выбрасывать большие фрагменты кода.

Исключение 1 — одноразовые программы

Такой код должен быть выполнен лишь однажды или несколько раз — например, в программе, обеспечивающей миграцию отдельного приложения, весь массив данных которого адаптируется к новой версии программы. Тогда код просто исправляют до тех пор, пока он не выполнит свою задачу. В таком случае перфекционизм не оправдан.

Исключение 2 — прототипы

Прототипы — это наброски, призванные представить идею в коде. При их разработке допустимо пренебрегать такими вещами, как обработка ошибок, реализуя только идею и не учитывая, как впоследствии может измениться цель ее применения. Есть хитрый прием: регистрировать в системе контроля версий и прототипы, таким образом, в качестве резервной копии сохраняется вся история разработки, а конкретный прототип может содержать интересные идеи, которые впоследствии также могут пригодиться.

К сожалению, в таком случае прототип зачастую рассматривается как первая версия приложения. Именно в данной ситуации вам аукнутся все жестко закодированные значения и подавленные исключения. Эту информацию можно рационально использовать, занимаясь рефакторингом, модуляризацией и отладкой кода.

Вполне можно выстроить на основе прототипа серьезную систему, если найдешь время еще раз обдумать кое-как склепанную архитектуру, причесать мелкие и не очень мелкие огрехи, а также позаботиться о надежности, удобстве поддержки и расширяемости системы. Во многих компаниях с этим возникают проблемы, поскольку отдельное время на такую работу не выделяется.

Исключение 3 — смутно представляешь себе проблему

Случается, что ты хочешь что-то реализовать, но сам не вполне понимаешь, что именно. В таких случаях лучше сначала написать код и посмотреть, что получится, чтобы полностью представить себе проблему. Затем код меняют, причесывают неаккуратные места и обеспечивают правильную обработку ошибок.

Такой способ итерационной разработки кода очень действенный, если применять его в небольших объемах, например, чтобы освоиться с незнакомой библиотекой или протестировать решение задачи. Однако такая практика дает прямо противоположный результат, если внедрить ее в рамках большого проекта, поскольку неаккуратные приемы навсегда укореняются в коде.

В определенных объемах допустим и копипаст кода, то есть использование небольших фрагментов, надерганных с форумов по программированию или из блогов. Такие списанные кусочки кода зачастую не слишком подходят к конкретной программе, могут быть устаревшими или некачественными. Но преимущество фрагментов в том, что они работают и могут пригодиться в другом месте — там, где в противном случае пришлось бы часами читать документацию и разбираться в концепциях. Если вы слепо копируете где-либо код, то должны хотя бы прочитать, что еще пишут о нем в данном источнике (например, в качестве комментария), а также оставить соответствующие комментарии в своем коде.

Расстановка приоритетов

После перерыва в несколько лет я захотел познакомиться с новыми веб-технологиями, в частности лучше разобраться с объектной моделью JavaScript, подробнее изучить jQuery и опробовать новые возможности HTML5. Для этого

я в свободное время написал своеобразный конструктор коллажей с веб-поддержкой. В этом приложении можно комбинировать имеющиеся картинки и снабжать их текстом. Реализовать этот проект я решил за четыре недели — у меня как раз выдался декретный отпуск, — собираясь к концу этого срока запустить работоспособный прототип. Это небольшой срок, учитывая, что за это время требовалось охватить три большие темы, попутно заботясь о младенце. Поэтому я решил, что объектная модель будет иметь низкий приоритет и я не буду разбивать код на модули, как предполагал заранее. Вместо этого я решил все записать в огромный файл JavaScript в надежде, что позже у меня будет время причесать этот код. Пришел к такому выводу, поразмыслив и решив: в столь небольшом проекте недостатки, связанные с отсутствием модульности, будут не слишком серьезны. К тому же у меня было ощущение, что если я возьмусь за членение объектов без понимания объектной модели, то мне придется еще два или три раза переписывать все заново.

На этот раз стратегия сработала. Сейчас у меня есть рабочая система, я хорошо представляю себе HTML5 и jQuery, а в настоящее время занимаюсь модуляризацией кода.

Йоханнес

Исключение 4 — альтернатива была чересчур затратной

Этот случай деликатный, поскольку подталкивает к ложным оценкам ради удобства. Зачастую, если сложность кажется непреодолимой, это также признак того, что либо вы не вполне поняли задачу, либо имеющийся код плохо структурирован.

Но все-таки во всех случаях есть о чем поразмыслить: о миграционных сценариях, о том, как переправить информацию из одной базы данных в другую, причем запуск и тестирование требуется выполнить на одной машине, вплоть до руководства по восстановлению, в ходе которого должны экономиться до трех уровней вложения.

Исключение 5 — на вас внезапно спускают дедлайн

Как правило, гибкие сроки могут лишь удлиняться — по крайней мере мы любим себя в этом убеждать. Но иногда дедлайн возникает на горизонте совершенно неожиданно. Это особенно неприятно в тех случаях, когда ты только начал разработку какой-то важной возможности, понимаешь, что она и близко не готова, но заказчик требует работоспособный прототип «как можно скорее, лучше — завтра».

Или — реальная история — ты разрабатываешь приложение для Android просто из любви к искусству, чтобы похвастаться, что именно ты написал первое приложение такого рода, а потом случается где-то прочесть расплывчатые сведения о том, что кто-то другой уже пишет практически то же самое. Ты чувствуешь досаду, словно проигрываешь гонку к Южному полюсу, идя вторым, поэтому наскоро делаешь минимальную версию и надеешься, что в нее не вкрадутся совсем уж одиозные баги. Позже обязательно найдется время, чтобы препарировать код с техническим долгом и заново его собрать.

Ситуации, в которых неаккуратные приемы и халтура наверняка вам аукнутся, — когда вы наскоро делаете заплатки для функционирующих систем. Если где-то что-то не работает, то практически никогда не стоит «быстренько искать» ошибку и исправлять ее. Во-первых, в условиях цейтнота сам цейтнот провоцирует ошибки. Во-вторых, поскольку вы допустили исходную ошибку, а не подумали над задачей подольше, ошибочно полагать, что сможете заново вникнуть в задачу и решить ее лучше, чем в прошлый раз.

Исключение 6 — вы хотите победить в «конкурсе по обфускации кода»

Некоторые разработчики умеют программировать так хорошо, что у них получается грозный, непонятный, но предельно выверенный код. Тогда они устраивают дуэли с такими же профессионалами. Цель таких «конкурсов по обфускации кода» — написать программу, которая в идеале будет понятна лишь машине, но не человеку.

У начинающего программиста едва ли будет даже призрачный шанс победить в таком состязании, даже если он ничего не понимает в коде и по невежеству игнорирует всяческие соглашения. Призовой код в таких соревнованиях непонятен, но сделан таким специально, поскольку вышел из-под руки мастера.

Если вы все-таки хотите попробовать свои силы в таком конкурсе, то вам сюда: https://en.wikipedia.org/wiki/Obfuscation_(software), выберите конкурс на том языке, который вам нравится.

Можете ссылаться на нас и в следующих случаях, когда хотите применять небезупречные методы.

- Все последствия хорошо обозримы, поскольку речь идет о небольшом проекте или об отдельном модуле в более крупном проекте (разумеется, если он не связан с банковскими счетами или чьим-нибудь здоровьем).
- Вы сознательно ломитесь через свой проект «в грязных сапогах» и намереваетесь позже навести в нем порядок.
- Позже у вас найдется время и возможность устранить устроенный вами хаос.

В идеале следует четко помечать плохой код как плохой.

Разумеется, с этим списком связана еще одна проблема: можно практически в любой ситуации чистосердечно утверждать, что в ней актуально хотя бы одно из описанных условий. В таком случае этот список не работает в качестве разграничительного инструмента. Возможно, продолжая приведенный ранее пример с долгом, следует сказать о некой внутренней «картотеке должников», которая велась бы в «Обществе защиты высокого качества кода». Всякий раз, когда берешь технический кредит, в этой картотеке делается новая запись. Кредиты погашаются при рефакторинге плохого кода, в таком случае запись удаляется. Пока есть непогашенные выплаты, новых кредитов не берем. Таким образом удается хотя бы однажды избежать в коде ошибок, связанных с желанием сделать по-быстрому, при условии, что придерживаешься правил такого вымышленного общества.

Если вы так никогда и не соберетесь с духом, чтобы прибраться в плохом коде, всегда остается возможность купить своеобразную индульгенцию. На сайте codeoffsets.com можно избавиться от огрехов, допущенных в коде, подобно тому как на других ресурсах можно заплатить за лишние выхлопы. Так, три некачественные строки вам там исправят за $1,5, плохо спроектированный класс — за $50, а полная переработка приложения потянет на $5000. Эти деньги идут на поддержку различных свободных проектов, посвященных борьбе против плохого кода.

ЧАСТЬ IV

Выбор средств

19 Не делай сам

> Однажды жил в Польше бедный еврей, у которого совсем не было денег на учебу, но он питал особую страсть к математическим наукам. Он читал, читал то, что мог приобрести, — это была пара дешевых книжек. Он думал, и учился, и еще раз думал. И в один прекрасный день он совершил Открытие, он обнаружил целую Систему и почувствовал: я что-то сделал. А когда он покинул свой маленький городок, он увидел новые книги, и все, что он сделал, все, что он открыл, уже было здесь — это называлось дифференциальным исчислением. Тогда он умер. Люди говорят: чахотка. Но он умер не от нее.
>
> *Курт Тухольский. Не бывает нетронутого снега*

Неопытные программисты тратят много времени, заново изобретая функции, которые уже имеются в используемом ими языке или его стандартных библиотеках. Разумеется, на первых порах невозможно получить представление обо всех функциях конкретного языка программирования. Никто не будет начинать изучение языка, запоминая все его команды в алфавитном порядке[1]. Тем не менее как же без особых усилий определить, где целесообразно корпеть над задачей и программировать самому, а где вы всего лишь заново изобретаете велосипед (с треугольными колесами)?

> Вот неполный список команд PHP, которые я по незнанию написала сама: array_rand, disk_free_space, file_get_contents, file_put_contents, filter_var, htmlspecialchars, import_request_variables, localeconv, number_format, parse_url, strip_tags, wordwrap.
>
> *Катрин*

Однако неумелые программисты не пользуются имеющимися решениями не только по неопытности или незнанию. Даже если поставленная задача вызывает у программиста робкое подозрение о том, что кто-то уже мог ранее с ней столкнуться и найти решение, далеко не всегда делается следующий логичный шаг: найду-ка

---

[1] Почти никто. В принципе, это очень хорошая идея, но приступать к чтению такого списка команд следует уже на том этапе, когда подыскиваешь какие-нибудь экзотические функции, о существовании которых в свое время и мечтать не решался.

я это решение. Многие воспринимают программирование как интересное и увлекательное занятие, поэтому гораздо охотнее пишут код, чем ищут уже написанный. Кроме того, люди склонны переоценивать собственные способности, а также недооценивать сложность стоящей перед ними задачи. Как раз те задачи, которые кажутся вполне обозримыми, оборачиваются проблемами, хотя неопытный программист, ознакомившись с задачей, почти сразу полагает, что нашел если не решение, то путь к нему.

Внимательные читатели распознают в этих проблемах оборотную сторону важных программерских добродетелей, перечисленных Ларри Уоллсом (о них мы говорили в главе 2): лени, нетерпения и переоценки собственных сил. Эти качества программиста превращаются в достоинства лишь тогда, когда специалист начинает использовать их к месту. Тот, кто самостоятельно пишет все свои инструменты, не озаботившись предварительным исследованием, упускает время, которое можно было бы потратить на создание чего-то действительно полезного, еще не существующего. Ведь первая рабочая версия собственной функции расчета даты, движка для блога или инструмента для учета рабочего времени пишется на удивление быстро, но стоит только начать ею пользоваться, как поразительно быстро всплывают какие-то частные случаи, баги, пожелания по расширению... А когда с кодом начинают работать другие люди, его поддержка может отнять у вас огромнейший кусок жизни. Согласно известному эмпирическому правилу опытный программист с учетом обдумывания, тестирования, отладки, оптимизации и документирования пишет примерно десять строк отточенного кода в день (кстати, у писателей схожая ситуация). Неопытный разработчик, вероятно, успеет не больше.

Для решения постоянно повторяющихся проблем существуют стандартные решения, доведенные до совершенства многими поколениями программистов. Таковы, например, алгоритмы сортировки. Необходимость работы над задачей, требующей особого, ранее не существовавшего принципа сортировки, сама по себе маловероятна для среднего программиста. Если предлагаемое во фреймворке стандартное решение не является идеально подогнанным под конкретную ситуацию, это еще не повод разрабатывать собственное решение, поскольку в таком случае вы не только тратите время, но и, вероятно, допускаете ошибки, а также работаете вразрез с фреймворком. Иначе говоря, в итоге вы реализуете для себя прекрасный алгоритм сортировки, но нестыковки между кодом и фреймворком проявятся где-нибудь в другом месте.

Применять готовые решения целесообразно и потому, что самодельные инструменты не обрадуют тех, кому в будущем придется работать с вашим кодом и читать его. Возможно, эти специалисты захотят повысить производительность приложения или найти в нем ошибки. Если же в нем вместо вызова проверенных стандартных функций всплывет ваше собственное решение, то незнакомый код вызовет справедливое недоверие. Читателю кода придется самостоятельно разбираться с деталями странной функции сортировки, чтобы убедиться, что она не содержит ошибок и действительно эффективно решает задачу. Напротив, если бы в коде была использована стандартная функция, то опытный читатель мог бы всем этим не заниматься, поскольку реализация стандартной функции проверена уже столько раз, что можно не сомневаться: ошибок в ней нет и работает она эффективно.

Что делать

Если ловишь себя на том, что снова и снова пишешь похожий код для решения относительно простой задачи, скорее всего, для такого случая уже есть готовое компактное решение. Существует ряд возможностей нащупать такое решение.

- Читаешь документацию по языку, используемому в данной предметной области. Может быть, удастся найти перекрестные ссылки на нужное решение.
- Смотришь список всех функций или функций из определенной тематической области и надеешься, что нужная функция будет иметь говорящее название. Так, функция `array_rand` из приведенного ранее примера легко отыскивается в документации по языку PHP на сайте `php.net` в разделе «Функции массивов».
- Сверяешься с книгой, в которой перечисляются стандартные решения. Для этого хорошо подходят сборники рецептов от издательства O'Reilly (в оригинале такие книги называются Cookbook). В них формулируются типичные вопросы, возникающие при программировании, ответы на которые даются в форме рецептов.
- Забиваешь проблему в поисковик и ищешь решение на таких сервисах, как stackoverflow.com, где сразу множество пользователей предлагают решения одно элегантнее другого. Например, я наткнулся на функцию `array_rand`, задав запрос `"how to" "random element" array php`.
- Заходишь на сайты github.com или sourceforge.net и ищешь в описаниях свободных проектов написанный на интересующем вас языке. Существует огромная вероятность того, что ваша проблема уже решалась. Затем выясняешь, как авторы проекта справились с задачей.
- Когда накопишь некоторый опыт, то и в сравнительно новом языке представляешь, какие функции есть в распоряжении. Остается уточнить их названия и технические детали. Уже не придется формулировать в поисковике какой-нибудь тяжеловесный запрос вроде `"how to" "random element" array php`. Вместо этого достаточно спросить: `array_rand in python`, чтобы найти питонский эквивалент уже знакомой команды PHP.

Лишь окончательно убедившись, что нигде нет готового кода, которым можно было бы воспользоваться, стоит приступать к программированию самому. При этом может очень пригодиться и исходный код, написанный на другом языке, — с его помощью вы сможете как минимум оценить истинные масштабы стоящей перед вами задачи.

Больше слушай, меньше работай

Я как раз учусь не заниматься ненужной работой: постоянно слушаю подкасты о свободных проектах. Причем я учусь даже на тех, названия которых на первый взгляд меня совершенно не интересуют. Чтобы не изобретать велосипед, сначала нужно узнать, что велосипед существует. Здесь очень помогает знакомство

с ландшафтом свободных разработок. Кроме того, эти подкасты избавляют от предубеждения "весь этот Open Source нерабочий, корявый и отвратительный", которое, по существу, является очередной ипостасью неприятия чужих разработок (принцип ЭНМП — "это не мы придумали"), — достаточно послушать, с каким запалом основатели проектов рассказывают о своих детищах.

Англоязычный подкаст FLOSS weekly (twit.tv/floss) регулярно представляет те или иные свободные проекты. В этом подкасте основатели могут рассказать, что подвигло их на конкретную разработку, какие решения они приняли при ее проектировании. В других выпусках тема обсуждается шире. Очень рекомендую программистам, впервые берущимся за новый язык, библиотеку или технологию, пару часов послушать такие подкасты.

Ян Бёльше

Библиотеки

Если вам не удалось найти в рассматриваемом языке такую функцию, которая предназначена для решения конкретной задачи, то, возможно, она действительно отсутствует в ядре языка. При решении относительно сложных задач (соединение с базой данных, синтаксический разбор XML, обработка изображений или лингвистические операции), то по умолчанию в вашем языке, вероятно, не будет готовых инструментов для этого. В таком случае придется поискать решение в соответствующих библиотеках.

Библиотека[1] — это подборка функций (либо классов, если речь идет об объектно-ориентированном языке), предназначенных для решения схожих проблем, возникающих в различных программах. (Да, это касается поразительно большого количества проблем.) В идеальном случае автор библиотеки особенно старается сделать интерфейс своего библиотечного кода максимально понятным широкой аудитории и хорошо его документировать. Этот интерфейс библиотеки называется API (интерфейс программирования приложений). Создавая хороший API, его разработчики уделяют особое внимание строгому соблюдению соглашений (см. главу 4), чтобы сделать интерфейс предсказуемым и тем самым облегчить жизнь разработчикам приложений.

Языки программирования предоставляются разработчику не в голом виде, а содержат очень похожие наборы функций, иначе было бы проблематично написать какую-либо осмысленную программу. Подобный набор называется стандартной библиотекой или библиотекой времени исполнения[2]. Самый приятный аспект стандартных библиотек заключается в том, что программист просто подразумевает, что она существует, и рассчитывает, что сможет использовать ее

---

1 Термины «библиотека» и «пакет» здесь употребляются как синонимы.

2 Как правило, в таких библиотеках содержатся основополагающие функции для операций с базами данных, работы с текстовым вводом/выводом, математические функции, функции для обращения со списками, массивами и строками.

функции столь же безусловно, как и команды, встроенные в язык. Часто далеко не просто провести границу между ядром языка и стандартной библиотекой, но на практике это редко имеет значение.

Иная ситуация складывается с библиотеками, которые не так прочно связаны с языком. Ведь такие библиотеки программисту для начала приходится отыскать, потом скачать и установить. Эти специальные библиотеки должны присутствовать и на компьютере конечного пользователя, в противном случае ему придется удовлетвориться веб-приложениями, для работы которых нужен лишь браузер. Если вы хотите применять такие библиотеки, то должны обдумать, как они попадут на ваш компьютер и при необходимости на компьютеры ваших пользователей. (Для этого обычно также существуют готовые решения, их и нужно применять, а не выдумывать что-либо самостоятельно.)

Ситуация может стать по-настоящему сложной и нервозной, если для работы с библиотекой требуются три другие библиотеки, а для работы этих трех — еще пять. Подобные зависимости могут настолько усложниться, что с ними едва можно управиться. Из-за этого не слишком искушенные программисты однажды могут в сердцах все бросить и решить: «А сделаю-ка я все сам». Программы, помогающие разрешать зависимости и устанавливать библиотеки, называются *менеджерами пакетов* (им будет посвящен специальный раздел в главе 20).

Если в том языке, на котором вы работаете, существует сообщество программистов, организовавших такую пакетную установку библиотек, причем сообщество ведет соответствующий онлайновый каталог, — вам повезло. Ведь именно в таких каталогах удобнее всего искать конкретную библиотеку для решения стоящей перед вами задачи. Поэтому к ним следует обращаться в первую очередь. В некоторых из этих каталогов можно подписаться на RSS-ленты, в которых сообщается о новых поступлениях, в других есть система оценок и другие механизмы обеспечения качества. Некоторые из этих источников перечислены в табл. 19.1.

Таблица 19.1. Каталоги библиотек

Язык	Библиотечный каталог	Адрес в Интернете
JavaScript	JavaScript Libraries	javascriptlibraries.com
Python	Python Package Index	pypi.python.org
Perl	Comprehensive Perl Archive Network	cpan.org
R	Comprehensive R Archive Network	cran.r-project.org
Ruby	Ruby Gems	rubygems.org
PHP	PECL (PHP Extension Community Library)	pecl.php.net
PHP	PEAR (PHP Extension and Application Repository)	pear.php.net
Node.js	Node Packaged Modules	npmjs.org

В Java и C++ такие удобные каталоги, к сожалению, отсутствуют. По адресу javascriptlibraries.com находятся только важнейшие библиотеки. Однако программистам на C++ следует обратить внимание на коллекцию библиотек boost.org. Этот сайт не слишком доступен для неопытных программистов, поскольку сам по себе сложен, но программисты C++ привыкли к таким проблемам.

Кроме этих каталогов, связанных с конкретными языками, есть интернет-сервисы вроде github.com, bitbucket.org и sourceforge.net, на серверах которых можно скачать всевозможные свободные проекты, в частности, их код.

Привязки к языкам

Зачастую различные решения, специфичные для конкретных языков, базируются на одной и той же библиотеке, которая обычно написана на C. В таком случае библиотека состоит из небольшого «перевода», позволяющего применять ее с разными языками. Если вы слышите фразу: «Мы используем PHP-привязку для работы с open-ssl», это означает, что в проекте применяется криптобиблиотека open-ssl, написанная на C, внедряемая в экосистему PHP с помощью соответствующего PHP-модуля.

Практически во всех основных сферах задач уже есть бесплатные библиотеки, которые легко найти. Обычно даже не одна, а от двух до десяти. Программист оказывается перед выбором, и на выбор требуется время. Железное правило: брать наиболее распространенную библиотеку, или ту, сайт которой поддерживается лучше всего (это признак того, что разработчики заботятся о ее доступности), или ту, в которой самая лучшая документация и наиболее понятный API.

Также обращайте внимание на лицензию. Есть такие лицензии, как GPL, они обязывают вас предоставлять весь свой программный код по такой же лицензии, если вы употребите в нем хотя бы небольшой вставной фрагмент, подпадающий под лицензию GPL. Лицензия Afferto-GPL (AGPL) идет на шаг дальше и требует публиковать весь код вашего веб-сервиса, если вы применяете в нем AGPL-лицензируемые библиотеки. Правда, обычно библиотеки предоставляются по менее строгим лицензиям, как то: BSD, MIT или LGPL. Но все-таки уточняйте такие вопросы, если разрабатываете не свободный проект.

Обращение с чужим кодом

Если вы скопируете фрагмент кода либо напишете собственный код на основе чужого, который где-нибудь видели, то получившийся код окажется неоднородным. В принципе, это неизбежно, неопытным программистам приходится с этим смириться, просто нужно пройти этот этап, осваивая искусство программирования. Лучше всего просто признать этот факт и позаботиться о том, чтобы границы между собственным и чужим кодом были максимально заметны, при этом чужой код выносится в отдельные файлы. Но если вы старательно подгоняете чужой код под собственные соглашения (см. главу 4), то столкнетесь со сплошными неудобствами. Во-первых, ситуация, вероятно, только ухудшится, во-вторых, вы будете тратить время впустую, в-третьих, получившийся код не удастся поддерживать, если чужой код в источнике впоследствии будет обновлен. Вместо этого можно рассмотреть такой вариант: сложные вещи запрятать поглубже, а повыше разместить собственную функцию на основе вызова библиотечной — такой, у которой гораздо

проще список параметров и опущен весь функционал, который не требуется в вашем проекте. Такой код называется *оберткой*.

Небольшие фрагменты чужого кода можно брать когда вздумается, даже если вы не вполне их понимаете. Это один из самых простых и красивых способов обучения. Однако нужно иметь хотя бы минимальное представление о том, что делает чужой код. В противном случае велика опасность того, что он будет не столько решать задачу, сколько вызывать совсем иные побочные эффекты. При обширных заимствованиях полезно оставлять в качестве комментария гиперссылку на источник. Тогда позже там можно будет еще раз посмотреть пояснения, а иногда даже найти обновления. Кроме того, такая практика позволяет предупредить упреки: «Вы присваиваете чужое». Помечать источник — хорошая идея, даже если это не требуется по лицензии.

Популярные ошибки при работе с чужим кодом. Не изменяйте ничего в локальной копии библиотечного кода. После этого чужой код будет уже невозможно обновить. То, что у вас получится, называется «ветка» (fork) — пусть даже вы ничего не слышали о ветках и не желаете о них знать.

ВЕТКА

Ветка (буквально — вилка) — это программный проект, возникший в результате изменений, которые когда-то были внесены в исходный проект. Поскольку эти изменения не учитываются при дальнейшей разработке исходного проекта (это может быть вызвано многочисленными причинами социального, политического или технического характера), оба проекта со временем все сильнее отдаляются друг от друга и стремятся к различным целям. При этом становится все сложнее переносить сделанные улучшения из одного проекта в другой или параллельно устранять найденные ошибки. Пример — различные производные операционной системы BSD: OpenBSD, FreeBSD, NetBSD.

Не используйте чужой код тайком, если автор открыто требует ссылаться на себя. Вы можете попытаться сделать это по различным причинам — из эстетических соображений («Ну и где же я здесь поставлю его имя?»), из личного тщеславия («Люди подумают, что я сам не разобрался») либо по желанию заказчика. Однако единственное верное решение таково: если вы не хотите или не можете указать автора, то не используйте такой код.

Не применяйте чужой код никогда, кроме случаев, описанных далее. Если вы заимствуете чужой код, проверьте, по какой лицензии он предоставляется, и убедитесь, что соблюдаете все условия. Все остальное — это нарушение авторского права, за что юридически иногда налагаются поистине драконовские наказания. Кроме того, чужой код вам не принадлежит.

Чего не нужно делать самостоятельно

Следующие задачи не нужно решать самостоятельно. Впрочем, это не возбраняется, но так вы просто немного усложните себе жизнь.

Функция, преобразующая буквы из строчных в прописные

Как минимум для английского и немецкого языка такая функция найдется. Возможно, она даже будет учитывать особое свойство буквы «ß», а именно удлинение строки на один символ, когда эта буква превращается в «ss». Подобные функции

существуют и для многих других языков. Сейчас вам кажется, что другие языки вам не понадобятся, поэтому вы можете удивиться, когда порой увидите в каких-нибудь именах собственных странные специальные символы.

Поисковая функция для собственного сайта

Поиск нужно писать быстро. Чтобы обрабатывался ввод в распространенных форматах, а вывод предоставлялся пользователю в удобочитаемой форме, требуется выполнить массу работы, на которую зачастую не отвлекаются. В крупных поисковиках даются указания, как без особых усилий встроить в собственный сайт, скажем, Google-поиск. Для большинства других практических случаев также есть готовые решения. Так, Lucene и Lucy — свободные проекты, оба от компании Apache Foundation, представляют собой библиотеки с функциями полнотекстового поиска для языков Java, C, Perl и Ruby.

Собственные синтаксические анализаторы для форматов файлов или URL

«Значения, разделенные запятыми, разве с ними возможны сложности? Это же просто значения, между которыми стоят запятые», — думаете вы и удаляете запятые с помощью `String.Split(',')` или `explode(",",$string)`. Но на самом деле даже со столь простыми форматами, как CSV и JSON, поразительно легко нажить себе проблемы: запятые могут быть экранированы либо стоять внутри цитат, то же касается кавычек. Значения могут распространяться на несколько строк... что уж говорить о более сложных форматах, например об XML. По аналогичным причинам не следует вручную отфильтровывать и параметры из URL.

Верно и обратное: не пишите собственный сериализатор (функцию, преобразующую переменные объекты в такой вид, чтобы их можно было сохранять или пересылать). Пользуйтесь многочисленными готовыми решениями, тогда не придется беспокоиться об экранировании, правильной расстановке скобок и других обременительных деталях.

Коммуникация с базами данных SQL

Практически для любой пары «язык программирования — база данных» есть модули, сбрасывающие в базу данных состояние объекта или структуры либо позволяющие восстанавливать состояние, уже записанное в базу данных. Это библиотеки для так называемого объектно-реляционного отображения (ORM). Их основная идея заключается в том, что программист не должен сам заниматься созданием SQL-команд. Ведь на самом деле это, во-первых, сложнее, чем кажется, во-вторых, в каждой базе данных этот процесс немного различается, в-третьих, именно на этой почве возникает целый класс брешей в безопасности (этой проблеме посвящен раздел «Внедрение SQL-кода и XSS — угрозы в пользовательском контенте» главы 25).

Самостоятельное написание регулярных выражений для распространенных задач

Если возникает малейшее подозрение, что кто-то уже мог решить данную проблему, нужно применять готовое решение, тем самым экономя время на отладку. Помочь в этом вам может, например, сайт regexlib.com.

Написание функций, устанавливающих те или иные события на определенные отрезки дня, недели или времени года

Эта задача на первый взгляд также кажется простой, но довольно скоро выясняется, что подножки здесь буквально на каждом шагу. Поэтому дальновидные люди в таких случаях пользуются встроенными возможностями, имеющимися в любой операционной системе. В Windows вы найдете планировщик задач (Windows Task Scheduler), в Unix (соответственно, и в Mac) для этого предназначен специальный инструмент cron (подробнее о нем — в главе 22). Если в код действительно требуется встроить таймер, то и для этой цели есть библиотеки. Вы найдете их по запросу `scheduling library` плюс название интересующего вас языка программирования.

Нетривиальные математические и физические алгоритмы

Если вы не профессиональный математик или физик, то не следует браться за решение любых задач, существенно выходящих за пределы школьного курса. Такие алгоритмы не только тяжело написать без ошибок, но и сложно тестировать. Для этого есть библиотеки, которые были отрецензированы и исправлены отзывчивыми коллегами. Кроме того, предлагаемые в них решения, вероятно, многократно быстрее и эффективнее, чем ваша собственная версия. Если вы уже знаете, что в работе над проектом вам придется решать математические и статистические задачи, то воспользуйтесь соответствующими программами (например, SAS) или языком вроде R.

Писать инструмент, измеряющий скорость кода

Эта задача также сложнее, чем кажется на первый взгляд. Будете ли вы запускать другие фоновые процессы? Сколько процессорного времени компьютер выделит на выполнение? Эта тема особенно актуальна в тех случаях (но не только), когда вы эксплуатируете сервер совместно с другими пользователями. Во всех языках программирования найдутся готовые решения такого рода. В JavaScript это Firebug (для Firefox) и Speedtracer (для Chrome), в Java есть программа jvisualvm, в Python — скажем, модуль timeit, в Ruby применяется Benchmark. Ищите по запросу `timing code` (код для таймера) или `profiler` (профилировщик). Если вы работаете с базами данных, то должны были хотя бы однажды сталкиваться с `explain`. Эта команда SQL сообщает интересные внутрисистемные детали о том, как был выполнен запрос. Немного попрактиковавшись, вы сможете понять по ее результату, удалось ли базе данных оптимизировать запрос, можно ли было использовать индексы и как вы могли бы дополнительно ускорить этот запрос.

Писать систему шаблонов для создания сайта

Есть много готовых инструментов, которые можно найти по запросу `template engine` (шаблонизатор) с указанием конкретного языка.

Писать собственный текстовый редактор для работы с полями ввода в HTML либо WYSIWYG-редактор, работающий в браузере

Как всегда, в данном случае легко реализовать первые 80 % функционала. На оставшиеся 20 % (если говорить о WYSIWYG-редакторах) даже в опытных командах

программистов тратятся годы работы. Что касается полей ввода для HTML, работа просто не стоит затраченных усилий, учитывая, какое множество готовых решений для этого имеется.

Писать код, срабатывающий при наведении указателя на объект, и другие распространенные украшения для сайтов

Вполне вероятно, что 99 % того, что вы хотите получить от функций Ajax, уже существует в готовом обкатанном виде. В данном случае чужой код не только сэкономит вам время и силы, но и убережет от неблагосклонности поисковика Google. Мэтт Каттс, сотрудник Google, решительно не советует самостоятельно писать код, срабатывающий при наведении указателя, поскольку он может легко вызвать недоверие у алгоритмов Google, исключающих из индекса те страницы, на которых скрыт текст-спам. Хороший источник Ajax-примочек — сайт script.aculo.us.

Делать собственные инструменты интернационализации

Тексты на вашем сайте должны быть на разных языках. Такую задачу решали уже многие люди до вас, поэтому для данной цели есть готовые решения, в частности утилиты gettext с привязками ко всем распространенным языкам. Чтобы найти другие нужные библиотеки, поищите по запросам `интернационализация`, `локализация`, `i18n` плюс название интересующего вас языка.

Писать собственные инструменты журналирования

Речь идет как о журналировании ошибок, так и о регистрации обращений к сайтам, отслеживании действий пользователя и об интерпретации файлов журналов. log4j — это библиотека для журналирования (см. подраздел «Ведение журналов» раздела «Инструменты и стратегии диагностики» главы 13), переведенная и на другие языки, где называется так: log4js (JavaScript), log4r (Ruby), Log4net (.NET). В Python она была сразу интегрирована в язык и называется просто logging. Чтобы log4j и аналогичные пакеты могли анализировать файлы журналов, существует, например, программа Apache Chainsaw, разработка которой, к сожалению, остановилась. В принципе, короткие файлы логов можно читать и анализировать прямо в текстовом редакторе.

Изобретать собственный язык разметки текста

Если вам нужен обычный понятный неспециалисту формат текстового ввода, который позже планируется автоматически преобразовать, скажем, в HTML, то, пожалуйста, не выдумывайте сами, как обозначать в списках курсив или жирный шрифт. Для этого не существует общепринятых стандартов, есть лишь общепринятые решения для определенных предметных областей. Так, в вики-источниках действуют свои соглашения, на форумах применяется формат *BBCode*, возможно, вас также заинтересуют форматы *Markdown* и *Textile*. В статье «Википедии» «Язык разметки» дается их краткий обзор. Лучше всего пользоваться распространенным языком разметки: есть готовые библиотеки, обеспечивающие конвертацию из одних форматов в другие. Если вам через пару недель покажется, что HTML-ввод лучше представить в виде PDF, то переделывать придется совсем немного.

Придумывать собственные соглашения

Если вы работаете в команде, откажитесь от изобретения собственных стандартов в коде. В мире достаточно распространенных умолчаний, а профессиональное взаимодействие — и так непростой процесс, а если при этом еще и придется отстаивать свои нововведения... (см. также главу 4).

Чего ни в коем случае не следует делать самостоятельно

Далее перечислены плохие идеи, чреватые гораздо более тяжелыми последствиями, чем деяния, описанные в предыдущем разделе. Некоторые из них будут стоить немало нервов и денег не только вам, но и вашим пользователям.

Настраивать правила расчета даты

В месяцах разное количество дней. Существуют временные пояса. Летнее время. Нумерация календарных недель. Високосные годы. Високосные секунды! Многие люди уже изрядно поломали голову над этими проблемами, результат их работы зафиксирован в библиотеках. Пользуйтесь этим, иначе наживете себе проблемы, как Ян Бёльше (см. врезку «Не делай сам, а то завтра погода испортится!»). Если вы усвоите это правило, то уже немного превзойдете программистов Amazon и Apple. В Amazon те книги, чьи Kindle-версии вышли 29 февраля, считаются опубликованными 28 февраля, а продажи, запланированные на 29 февраля, регистрируются 1 марта. Time Machine от Apple сохраняет данные резервного копирования от 29 февраля, однако эта дата не отображается в вашем пользовательском интерфейсе.

Не делай сам, а то завтра погода испортится!

«Здесь уже три дня льет дождь». Голос руководителя проекта по телефону звучит как-то обеспокоенно. Он стоит под искусственным светодиодным небом и глядит на хмурые облака над собой, которые, однако, должны были рассеяться еще два дня назад вместе с настоящими облаками, заволакивавшими небо над зданием. Это должно было произойти полностью автоматически в соответствии с изменением текстового файла на сервере Берлинского метеорологического института.

Вот уже два года система функционировала совершенно безупречно: научный сотрудник института записывает в текстовый файл уровень облачности и вид осадков, мой компьютер каждые десять минут скачивает этот файл, разбирает его, а затем в зависимости от результирующего ввода на огромном дисплее под крышей зимнего сада могут появиться облака.

Слушая по телефону шум дождя, я просматриваю файл журналов приложения и убеждаюсь, что он собирается обратиться на сервер института за новыми метеоданными лишь примерно через два миллиарда лет.

При этом я был совершенно уверен, что класс обработки дат, написанный мною собственноручно, наверняка не содержит никаких ошибок, ведь он столько лет безупречно работал и поддерживался в порядке.

Ян Бёльше

Самостоятельная реализация криптографии

Это гарантированно плохая идея, даже если ваш план связан с «исключающим ИЛИ, в котором я упомяну имя моей кошки» (см. главу 25). Криптография, подобно атомной энергетике, такая сфера, где все нужно обязательно поручать экспертам, поскольку все менее искушенные люди явно не представляют всех последствий своей деятельности. Некоторые считают, что в случае с атомной энергетикой этого не представляют и эксперты. Помните о так называемом законе Шнайера, названном в честь консультанта по безопасности Брюса Шнайера: «Любой человек может придумать такую умную систему безопасности, что он или она не может представить себе способ взломать эту систему».

Речь о криптографии идет не только в тех случаях, когда вам платит за защиту информации какая-нибудь спецслужба, но и всякий раз, когда дело связано с паролями. В случае если вы думаете: «Ах, что секретного в этих данных, здесь можно обойтись без полноценной криптографии», представьте, что бы случилось, если бы вся информация вашего пользователя хранилась и передавалась в незашифрованном виде. Если эта мысль вам неприятна, то обеспечьте настоящую криптографию, половинчатого решения быть не может. В частности, это означает, что вы не будете собственноручно мастерить входные страницы. Во-первых, так надежнее, во-вторых, уже есть готовые решения, в-третьих, общение с забывчивыми пользователями будет не таким нервозным (см. врезку).

Пароль: Alzh31m3r

Поскольку мой блог Riesenmaschine вырос из небольшой новостной рубрики обычного сайта, доступ к которой имели всего три человека, и поскольку я понятия не имела об аутентификации, вся защита доступа обеспечивается с помощью обычного каталога с файлами .htaccess. Вероятно, у этого метода много недостатков, о которых я до сих пор не знаю, но один из таких недостатков кажется огромным даже мне: каждый из 50 человек, эпизодически получающих доступ к сайту, иногда забывает свой пароль, а на сайте, конечно, нет механизма, который позволял бы пользователю изменять свой пароль. Мне просто приходит электронное сообщение об этом, и поскольку я тоже не записываю себе эти пароли, я вынуждена выдумывать новые, которые обладатели также могут забыть уже через неделю.

Катрин

Разрабатывать собственную систему отслеживания ошибок (багов)

Пользователь Stackoverflow Константин Веретенников приводит следующую статистику по имеющимся системам такого рода (по состоянию на конец 2012 года):

- Trac: 44 000 строк кода, 10 человеко-лет, на разработку потрачено $577 003;
- Bugzilla: 54 000 строк кода, 13 человеко-лет, на разработку потрачено $714 437;
- Redmine: 171 000 строк кода, 44 человеко-года, на разработку потрачено $2 400 723;
- Mantic: 182 000 строк кода, 47 человеко-лет, на разработку потрачено $2 562 978.

Если вы хорошо знакомы с сильными и слабыми сторонами имеющихся систем и у вас есть пара десятилетий свободного времени, не поддавайтесь на наши уговоры и воплощайте свои планы. Разумеется, современные системы отслеживания ошибок далеки от совершенства. В противном случае смиритесь с тем, что эти системы функционально не покрывают 100 % ваших потенциальных потребностей, и подарите себе от 10 до 47 человеко-лет свободного времени.

Разрабатывать собственный вики-движок

См. пример с системой отслеживания ошибок.

Разрабатывать собственный софт для блога

См. пример с системой отслеживания ошибок.

Изобретать собственные форматы файлов

Когда не хочется разбираться с тонкостями XML и JSON, быстро приходит на ум именно такая идея. Сама по себе она уже большая халатность, однако в худшем случае новоиспеченный формат файлов будет так похож на свой прототип, что будущий читатель (а то и вы сами) может принять его за оригинал, а потом немало удивляться и рвать на себе волосы.

Писать код, проверяющий, является ли строка или URL адресом электронной почты

Если требуется это сделать, то нужны либо библиотека, либо обескураживающе длинное и сложное регулярное выражение. Если допустите ошибку, то лишь рассердите своих пользователей, чья почта будет отбрасывать совершенно корректные адреса. Имеющиеся решения эволюционировали как результат совместного труда многих людей, учитывают многочисленные шишки, набитые при решении нестандартных случаев. Вероятность того, что кто-то в одиночку повторит это достижение, сопоставима по вероятности с самозарождением броненосца из ила. В качестве иллюстрации приведем здесь пример такого решения:

```
\b(([\w-]+://?|www[.])[^\s()<>]+(?:\([\w\d]+\)|([^[:punct:]\s]|/)))
```

(Источник — http://daringfireball.net/2009/11/liberal_regex_for_matching_urls, объяснение и советы по возможной оптимизации выражения — http://alanstorm.com/url_regex_explained.)

А вот другой вариант решения:

```
/^(https?):\/\/((?:[a-z0-9.\-]|%[0-9A-F]{2}){3,})(?::(\d+))?((?:\/(?:[a-
z0-9\-._
~!$&'()+,;=:@]|%[0-9A-F]{2}))(?:\?((?:[a-z0-9\-._~!$&'()+,;=:\/?@]|%[0-
9A-F]{2})))?(?:
#((?:[a-z0-9\-._~!$&'()+,;=:\/?@]|%[0-9A-F]{2})*))?$/i
```

(http://snipplr.com/view/6889/regular-expressions-for-uri-validationparsing/.)

Еще удобнее применять готовые регулярные выражения. Потратьте пару минут на поиск по запросу email validation library (библиотека для валидации электронной почты) и укажите конкретный язык. Прибегайте к регулярным выражениям лишь в случае, если такой поиск не даст подходящего результата.

Удалять HTML-теги из строки с помощью регулярных выражений

Такой метод кажется невероятно привлекательным неопытным программистам. Но HTML сложнее, чем может показаться на первый взгляд, а для каждого рабочего регулярного выражения[1] существует как минимум один частный случай, на котором можно споткнуться, причем во время работы с абсолютно корректным HTML. Разве что Чак Норрис может разбирать HTML с помощью регулярных выражений. Всем остальным для этой цели рекомендуется «погуглить» подходящую библиотеку (запрос, например, HTML Parser, HTML Sanitizer, HTML Purifier или parse html library). Так экономится масса усилий, которые пришлось бы потратить как на придумывание, так и на многолетнее исправление сложных регулярных выражений[2].

Экранирующие процедуры, позволяющие удалять код SQL, JavaScript и XSS

Прежде всего (и не только в Web) бывает необходимо замаскировать управляющие символы языка, чтобы можно было обрабатывать фрагменты программы на этом языке. Если вы хотите опубликовать на сайте примеры из HTML, то не можете просто вставить этот HTML где заблагорассудится, а должны сначала экранировать символы < и >. Известно, как сложно экранировать активное содержимое, и в каждом языке есть функции или библиотеки для этой цели.

Формулировать собственные лицензии и договоры

Здесь также найдутся готовые решения, давно продуманные специалистами. Применяя их, вы избавите правообладателей от огромной работы (и, возможно, от расстройства). Поищите в Creative Commons подходящую лицензию, если хотите выкладывать в Интернете тексты или фотографии. Если нужно составить договор, обратитесь к юристу.

К сожалению, официальные и широко распространенные решения тоже не всегда верны. Поэтому все советы, приведенные в этой главе, касаются лишь неопытных или малоопытных программистов. Им просто: код другого человека с большой вероятностью окажется лучше, быстрее и корректнее, чем их собственный, его можно копировать без сомнений. Однако и начинающим порой целесообразно писать некоторые вещи самостоятельно при условии, что знаешь меру

---

[1] Например, это выражение может быть таким: <(?:"[^"]*"['"]*|'[^']*['"]*|[^'">])+>. Его автор, пользователь itsadok с сайта Stack Overflow, предупреждает: «Это регулярное выражение не учитывает блоков CDATA, комментариев, элементов для скриптов и стилей. Теперь хорошая новость: все это можно удалить с помощью регулярного выражения».

[2] Классический текст по теме — http://blog.codinghorror.com/parsing-html-the-cthulhu-way/ (перевод на русский язык — http://gautama-it.blogspot.com.by/2011/02/html.html).

и понимаешь, что делаешь. К сожалению, это бывает нечасто. Сеть изобилует дискуссиями о том, когда стоит прибегать к собственным решениям, а не полагаться на имеющиеся инструменты и библиотеки. Кроме того, вы найдете массу свидетельств опытных программистов, которые уже не раз и не два решались написать что-либо самостоятельно, а потом приходили к выводу, что их вполне устроило бы готовое решение, применив которое можно было бы сэкономить много сил и времени. Итак, речь идет о таком опыте, который каждый должен приобретать сам. Если вы проигнорируете наши советы и решитесь писать самостоятельно, то хотя бы будете заранее знать, каковы основные сложности, связанные с интернационализацией и функцией расчета даты. Вам будет проще — спустя недели, месяцы или годы, когда вы все-таки решите прибегнуть к готовому решению — оценить проблему и понять чужой код. Вы научитесь уважать работу и размышления, заложенные в чужой вариант решения.

Три решения одной проблемы

Катрин: «Когда-то я была человеком, написавшим движок для блога, сохраняющий отдельные записи в виде XML-файлов (примерно с таким обоснованием: “При необходимости я легко смогу изменить текстовый файл, а вот запись в базе данных — нет”). Для работы с этими файлами я написала XML-парсер, который при чтении сразу выбрасывает все, что стоит в угловых скобках, а при записи вновь заключает строки в угловые скобки. В коде стоит комментарий: “Потому что парсер для PHP никуда не годится”. Сегодня меня тянет добавить: “Здесь автор ошибается”. Даже если PHP-парсер действительно был так плох, я вряд ли была в состоянии это диагностировать».

Йоханнес: «Во времена, когда Интернет еще называли информационным хайвеем, а “Википедия” еще не была изобретена, я принял проект онлайновой энциклопедии, которую планировал доработать как в техническом, так и в содержательном отношении. Первая версия представляла собой программу на C, без веб-интерфейса. Она переводила текстовые файлы в HTML с помощью пары TeX-подобных подсказок.

Поэтому, принимая сайт, я сталкивался с определенными проблемами: нужно было быстро ввести `configure && make`, чтобы на сервере запустилась программа-конвертер. К сожалению, в результате я получал пренеприятнейший список ошибок, так как различные библиотеки были несовместимы.

Поскольку в последние пять лет мир не стоял на месте, у меня появилась отличная возможность не только заново изобрести конвертер, но и одновременно преобразовать данные в XML-описание. Чтобы ощутить, с каким энтузиазмом архитекторы послевоенной эпохи окончательно и бесповоротно ровняли с землей разбомбленные немецкие города, попробуйте выдумать XML-описание для хранения структурированных файлов. Великолепные неисписанные листы, никаких пут, связанных с обратной совместимостью, безграничный простор для творчества.

К сожалению, серверы того времени были недостаточно мощными для обработки крупных XML-файлов в реальном времени, а синтаксические анализаторы XML и близко не достигали нынешней степени оптимизации. Собираясь реализовать

веб-интерфейс, я сконцентрировался на том, чтобы преобразовать в HTML данные, содержащиеся в HTML, с помощью пары регулярных выражений так, чтобы пользовательский ввод можно было таким же образом расфасовать в угловые скобки. Получилось множество красивых и труднообнаружимых ошибок, из-за которых любой XML-анализатор брезгливо отвергал мои данные. Оказавшись перед выбором: написать полноценный анализатор и генератор XML либо как-нибудь локализовать проблему, я, к счастью, решил отсортировать ночью массив данных с помощью правильного парсера и с применением cron, чтобы отделить зерна от плевел. Найденные ошибки предстояло исправить вручную, что было не слишком профессионально, но хотя бы работало. А в дальнейшем оказалось, что собственноручно состряпанный XML оказался настолько удобоварим, что его даже можно было преобразовать в HTML или LaTeX для последующего изготовления PDF.

Оглядываясь назад, думаю, что, возможно, было бы быть легче и дальше применять простой формат данных первого поколения — из всех дополнительных возможностей, которые я внедрил, использовалась лишь малая толика».

Ян Бёльше: «Для проекта, в котором требовалось подружить два крайних, антагонистических сегмента IT-индустрии, а именно финансово-бухгалтерский софт и компьютерную игру, мне требовалась возможность сохранять выписки из счетов. Поскольку эта задача не критична по времени (доминирующую роль в процессе играет обмен информацией с банковским сервером, в данном случае это и есть узкое место) и поскольку на раннем этапе этого проекта я хотел иметь простой в обращении человекочитаемый формат, я остановился на XML. В свое время моим любимым языком был Python, так что я прочитал пару статей и постов в блогах о синтаксическом разборе XML в Python и ознакомился с API различных XML-модулей, поставлявшихся с дистрибутивом Python. В "Википедии" прочитал о коренных различиях между DOM- и SAX-парсерами. В конце концов я решил хранить данные за каждый месяц в отдельных XML-файлах. Поскольку эти файлы получатся не очень большими, я смогу одним махом прочитать их с помощью моего DOM-парсера, что явно проще, чем реагировать на акты разбора в SAX-парсере. Больше всего мне понравился API ElementTree, так как эта библиотека, пусть тогда и не входившая в состав дистрибутива Python, на мой взгляд, лучше всего решает основную задачу — выдавать простые решения для простых проблем. Например, список всех транзакций за месяц выводится с помощью всего двух строк:

```
import xml.etree.ElementTree as ET
transactions = ET.parse("transactions/2010-01.xml").getroot().fin-
dall("transaction")
```

Спустя пару версий Python ElementTree обрел статус стандартной библиотеки».

20 Инструментарий

> Я анализирую статистику публикаций в блоге Riesenmaschine с помощью программы для интерпретации астрономических данных, которая называется MIDAS (Munich Image and Data Analysis System). Кстати, с помощью этой программы я делаю и ВСЕ остальное. Не считая операций, которые можно выполнить в текстовом редакторе с помощью клавиатурного макроса, таких как интерпретация астрономических данных.
>
> *Алекс Шольц. Riesenmaschine*

Если не имеешь понятия о теме, то сложно подобрать для работы верный инструментарий. В программировании эта ситуация даже еще сложнее, чем в других сферах, если не считать, пожалуй, накопительного страхования жизни. В то же время подбор инструментов играет определяющую роль для комфортности программирования, а если говорить об опытных специалистах, то и для эффективности.

Многие бывалые программисты по фундаментализму и ортодоксальности могут потягаться с моджахедами. Они рьяно защищают любимые языки, текстовые редакторы и системы сборки. Если собрать вместе группу программистов и немного их подпоить, то можно наблюдать спонтанное зарождение бесконечного спора непримиримых оппонентов.

Это — вкупе с гордыней — связано с тем, что для большинства задач из области программирования не найдется такого языка или текстового редактора, который будет явно лучше всех остальных. У каждого инструмента есть свои достоинства и недостатки. Например, очень мощный редактор, вероятно, не удастся быстро освоить, а в интуитивно понятном не хватает многих возможностей. Но со временем, приспосабливаясь к своим инструментам, вы научитесь обходить либо игнорировать их слабости, а сильные стороны освоите в совершенстве.

Все это подводит начинающих программистов к успокаивающей мысли о том, что, избрав предметную область, легко влиться в ее мейнстрим — и уже будет какой-то результат. У большинства обкатанных инструментов найдется по паре настоящих достоинств, иначе эти инструменты не пользовались бы такой популярностью. Благодаря тому что они так распространены, удается быстрее решать возникающие вопросы и проблемы.

Редакторы

Пусть, в принципе, и возможно писать большие программы в Блокноте или с помощью SFTP-инструмента для командной строки, такое решение никуда не годится. Строго говоря, это даже не костыль, а ноющая рана.

Хотя программирование — относительно сложная высокотехнологичная деятельность, при которой не помешает некоторая техническая поддержка, часто можно прочитать о том, что все, кроме редактора и компилятора, — излишняя мишура. Возможно, для программиста с 20-летним опытом это действительно так, но начинающий не должен поддаваться на такую провокацию.

Вам в самом деле абсолютно, решительно не обойтись без редактора, который:

- может читать файлы UTF-8 и ISO-8859 и пересохранять их из первого во второй формат и наоборот. Кроме того, он должен принимать окончания строк для Windows и Unix;
- обладает языковым синтаксическим модулем, предназначенным для программирования. Такой синтаксический модуль окрашивает текст разными цветами в зависимости от того, что перед нами — комментарий, функция или синтаксическая ошибка;
- позволяет искать и заменять текст одновременно во многих файлах. Пожалуй, единственный редактор, не обладающий такой возможностью, — это Блокнот, поэтому наш однозначный совет: выясните, как в вашем редакторе выполняются одновременный поиск и замена в нескольких файлах. Кроме того, обращайте внимание и на предупреждения, связанные с глобальным поиском и заменой, которые перечислены в разделе «Среды разработки» данной главы и в главе 5.

Вероятно, даже у самого скромного редактора больше возможностей, чем вы сейчас используете. Не помешает время от времени открывать какие-то из них. Например, узнать:

- как создавать макросы для часто повторяющихся операций;
- есть ли несколько буферов обмена, которые избавили бы вас от постоянного переключения между двумя документами? Либо можно ли добавлять новые данные к тем, что уже имеются в буфере обмена;
- как заставить редактор отображать функции применяемого языка программирования. Иногда для этого требуются плагины (подключаемые модули);
- как включать регулярные выражения при поиске и замене;
- как заставить редактор подправлять плохо отформатированный код. По опыту, как раз при написании HTML или XML форматирование быстро сбивается.

Даже если прямо сейчас вы не выясните, как все это делается, полезно знать, обладает ли вообще редактор всеми этими возможностями. Когда-нибудь вы настолько отчаетесь или вдохновитесь, что решите подробнее разобраться в деталях. Правило трех (см. главу 14) и здесь вам пригодится: если в третий раз нужно что-то сделать вручную или специально усложнить себе задачу, то, пожалуй, стоит

потратить пару минут на исследования и отыскать возможности, которые, может быть, предусмотрены в редакторе как раз на такой случай.

В статье английской «Википедии» *Comparison of text editors* («Сравнение текстовых редакторов») есть различные обзорные таблицы, содержащие сведения о том, какие редакторы какими востребованными возможностями обладают. Те из них, которые приспособлены специально для работы с HTML и веб-разработки, находятся в разделе Comparison of HTML editors.

Преимущества некомпетентности

Еще десять лет назад я пользовалась редактором, с помощью макросов которого могла сделать все. Это был NoteTab под Windows. Купив первый Mac, я была вынуждена осваивать новый редактор, но ни TextMate, ни SublimeText и близко не дали мне возможностей, которые у меня были в Windows. С одной стороны, это неудобно, я из-за этого иногда раздражаюсь. С другой стороны, мои отточенные навыки работы с, прямо скажем, очень скромным языком макросов NoteTab долго удерживали меня от того, чтобы решать задачи не с помощью Блокнота, а, скажем, на языке Perl. Кстати, неосведомленность о возможностях собственного редактора зачастую подталкивает меня разработать более зрелое решение — например, что-нибудь с использованием регулярных выражений. Преимущество такого подхода заключается в том, что и работа пойдет быстрее, и навыки меня не подведут, если я вновь поменяю редактор или операционную систему.

Катрин

Какой язык программирования правильный?

Треть всех программистов знают минимум один язык программирования.

БерндЭкенфельс/@eckes, Twitter, 26 сентября 2012 года

Полагаем, что вы имеете хотя бы ориентировочное представление как минимум об одном языке программирования, иначе вы читали бы не эту книгу, а что-нибудь «для чайников». Но, возможно, вы уже несколько лет вынашиваете туманные планы изучить второй или даже третий язык. Даже если вы уже можете изъясняться на пяти языках программирования, почему бы не освоить шестой?

Многие останавливаются на первом или втором языке программирования и говорят, что этот язык вполне отвечает их планам. Такая лень связана, в частности, с одним феноменом, который был описан известным эссеистом и программистом Полом Грэмом на примере вымышленного языка «Блаб»: «Когда наш гипотетический блаб-программист смотрит вниз на континуум мощности языков, он знает, что смотрит вниз. Менее мощные, чем “Блаб”, языки явно менее мощны, так как в них нет некой особенности, к которой привык программист. Но когда он смотрит в другом направлении, вверх, он не осознает, что смотрит вверх. То, что он видит, —

это просто странные языки. Возможно, он считает их одинаковыми с “Блабом” по мощности, но со всяческими сложными штучками»[1].

Интервью с Лукасом и Матиасом

Лукас: Что мне действительно помогло, так это изучение новых языков.

Матиас: Когда достаточно поучишься, осваивать новые языки станет совсем легко. Как только поймешь основополагающие концепции, остальное уже не составляет труда. Остается синтаксис, который изучается очень быстро. Кроме того, все еще раз укладывается в голове, если ты решал одни и те же задачи на разных языках. Любой язык построен немного по-своему, и уже поэтому он стимулирует такие решения, которые ты впоследствии сможешь применить в твоем первом языке.

Лукас: Бывает и так, что ты находишь язык, который подходит тебе гораздо лучше. Если ты придаешь большое значение микроменеджменту, то, пожалуй, следовало бы выучить ассемблер.

Матиас: Когда уже долго программируешь на каком-либо языке, привыкаешь к определенным идиомам, усваиваешь, как делаются определенные вещи, например, синтаксический разбор строки. А потом, в новом языке, это уже не работает. Нужно снова размышлять, как сделать то или это…

Лукас: …при этом, вероятно, ловишь себя на мысли: «О, я это делал слишком громоздко либо шел по устаревшему пути». Или дозреваешь до того, чтобы что-нибудь почитать по данной проблеме.

Матиас: С другой стороны, ты, конечно, переносишь на новые языки те концепции, которые уже изучил в старых. Старая идиома при этом не забывается. Просто ты пополняешь свой инструментарий.

Катрин: Да я и при работе с единственным языком постоянно разучиваюсь делать те или иные вещи.

Матиас: Но тогда ты плотнее изучаешь базовые принципы работы, а не отвечаешь на вопрос «Как я сделаю конкретно вот это?». Конкретные приемы для того или иного языка я могу быстро выискать в Интернете. Помечать себе подобные вещи в моем случае бессмысленно, это пришлось бы делать для слишком многих языков.

В Сети полно более или менее серьезных перечней, в которых упоминаются конкретные достоинства и недостатки отдельных языков программирования. При этом нас тянет дополнять такие списки и неосторожно разбавлять их собственными предпочтениями и антипатиями. Но детали таких конкретных рекомендаций меняются каждые пару лет, и вы не многому научитесь, если мы вам просто посоветуем: «Бери Python — с ним не ошибешься!» (хотя с ним вы и в самом деле не ошибетесь).

IT-консультант и автор Джеффри Мур делит всех пользователей-технарей на пять групп.

- Новаторы. Те, кто изобретают технологии или очень близки к этому.
- Ранние последователи. Эксперты-технологи, хорошо улавливающие, куда ветер дует, и раньше всех осваивающие новые технологии.

---

[1] http://www.nsc.ru/win/elbib/data/publ_cat/1225.pdf.

- Раннее большинство и позднее большинство — те пользователи, из которых состоит мейнстрим. Технология, воспринятая этими группами, является состоявшейся.
- Увальни. Инертные потребители, остающиеся верными технологиям позавчерашнего дня.

Языки программирования по схожему принципу распространяются среди нескольких групп пользователей, в разной степени готовых на риск. Чем более опытен и технически заинтересован программист, тем вероятнее он окажется в числе ранних последователей языка. Здорово, если позже этот язык закрепится, поскольку в таком случае у раннего последователя будет большая фора. Если же язык не приживется, значит, ранний последователь промахнулся. Для любого, кто уже выучил много языков, это будет менее болезненно, чем для неопытного человека, вынужденного признать, что его любимый язык морально устарел.

Вы, как начинающий программист, скорее должны относиться к раннему большинству: с одной стороны, вы еще не написали на языке собственных библиотек или больших проектов, поэтому не привязаны к ним — а в больших компаниях такая привязка наблюдается часто. С другой стороны, на освоение нового языка вам требуется больше времени, чем эксперту, причем вы, вероятно, не столь хорошо представляете, какие из новых подходов самые многообещающие.

Поэтому вот вам несколько сравнительно неустаревающих советов о подыскивании второго — пятого языка.

Потенциал языка зависит не только от его возможностей, но и — особенно для новичков — от имеющейся документации и экосистемы. Поищите в Интернете *документацию* по языку. Обратите внимание на то, насколько вы сориентируетесь в том, что найдете. Некоторые языки исчерпывающе документированы, причем в расчете на новичков, по другим, в принципе, также имеется вся документация, но понять ее очень сложно. По новым языкам зачастую просто нет полноценной документации.

Поищите по имени языка, добавив слово `tutorial` (руководство). Опробуйте первые шаги этого руководства. При этом игнорируйте внутренний голос, которой при каждой подобной попытке нашептывает: «Здесь все иначе! Не хочу!»

Выясните, как давно существует этот язык. Если он моложе пяти лет, тогда для вас, вероятно, найдется слишком мало подходящих инструментов, слишком мало документации, мало готовых решений в Сети, а в языке будет слишком много багов. Сообщество пользователей будет состоять из честолюбивых программистов-экспериментаторов, которым порой не хватит терпения отвлекаться на ваши проблемы. Ответ на вопрос «Только у меня такая проблема?» зачастую окажется утвердительным. Если язык старше 30 лет, то вы также рискуете остаться наедине со своими проблемами, а форумы по этому языку окажутся заброшенными.

Есть ли программисты среди ваших друзей, соседей, коллег? Беритесь за язык, на котором работают ваши знакомые. Даже если язык сам по себе окажется неидеален для тех задач, которые вы перед собой ставите, многие недостатки компенсируются тем, что при возникновении проблем вам просто будет с кем посоветоваться.

Если ваши друзья — взыскательные программисты, обгоняющие вас по опыту работы на много лет и не желающие, чтобы их уличили в опытах с недавно появившимся языком, поинтересуйтесь, с каким языком они работали пять лет назад, и изучайте его.

Иногда язык программирования выбирается в зависимости от идеи, которую хочешь реализовать. Например, если собираешься разрабатывать под Android — учишь Java, под iOS — к примеру, Objective-C. Зачастую определяющее значение имеет широкая экосистема, как в случае веб-разработки. Влияние веб не ослабевает уже 20 лет, в нем развились языки HTML и JavaScript. И тот и другой сталкивались с конкурирующими идеями, но в силу огромного успеха веб эти идеи не прижились.

Иногда специальный инструмент, с помощью которого можно решающим образом продвинуть ваш проект, есть только в одном или двух языках. Если вы ловите себя на том, что обрабатываете статистику посещений блога с помощью программы для интерпретации астрономических данных, то вам, вероятно, стоит подумать об использовании какого-нибудь математически-статистического языка, например языка R, в который многие статистические функции просто встроены.

Разузнайте, с помощью какого языка другие программисты решали подобные проблемы. (И затем подумайте, нельзя ли вам просто перенять уже найденное ими готовое решение — см. главу 19.)

Возможно, вы ищете подходящий язык для проекта, над которым работаете совместно с другими программистами. Если вы не самый хороший разработчик в команде, то просто следуйте предложениям коллег и не жалуйтесь на то, что вам приходится переучиваться. Рассматривайте это не как подчинение несправедливому требованию, а как ценную возможность выучить новый язык среди людей, которые им уже владеют. Не протестуйте, основываясь на каких-то отрывочных сведениях о языке, предложенном коллегами («Да ну, Java же такая громоздкая!»), если на самом деле вам всего лишь лень в нем разбираться.

Если вы вообще не собираетесь работать над конкретным проектом, а просто так, для себя хотите научиться программировать получше, то вам стоит присмотреться к тем языкам, которые хорошо знакомят с каким-то определенным способом мышления: Ruby — отличается приятной гибкостью, Smalltalk — особенно элегантно воплощает идею объектно-ориентированного программирования, Scheme или какой-нибудь из диалектов Lisp — примеры функциональных языков. Или C, а может быть, даже язык ассемблера — они заставят вас столкнуться с такими основами программирования, как использование указателей и управление памятью.

Да, велик соблазн выбрать в качестве второго языка тот, который кажется в чем-то похожим на хорошо знакомый первый. Но если на первом месте для вас стоит не столько достижение быстрых результатов, сколько желание стать хорошим программистом, то вы быстрее продвинетесь вперед с таким вторым языком, который как можно более не похож на первый. Если вы до сих пор работали с PHP или Ruby, познакомьтесь с C, C++ или Java. Если вы фанат объектно-ориентированного программирования, то Lua или R послужат для вас интересным контрастом. Если вам лучше знакомы языки с богатыми традициями, такие как Python или Perl, то, возможно, не помешает попробовать какие-нибудь совсем новые языки, такие как Go или Rust.

Иногда играет роль крутизна. Если вам (хоть вы и держите это в тайне) важно, чтобы, разговаривая с вами, другие программисты (пусть даже тайно) восхищались вами из-за языка программирования, который вы используете, и если это восхищение усиливает вашу мотивацию, то какой-нибудь только что придуманный язык, несмотря на свои недостатки, может оказаться как раз тем, что вам нужно. Иногда старые языки тоже обладают определенным фактором крутизны — например, язык ассемблера или Lisp. Но знание таких языков украсит ваш послужной список только в том случае, если вы еще довольно молоды. Иначе легко появляется подозрение, что вы выучили этот язык давным-давно, еще когда учились в институте благородных девиц, и с тех пор дальше не развивались.

Не обращайте внимания на таблицы, в которых указано, сколько строк требуется в разных языках на выполнение одной и той же задачи. Как новичок, вы должны радоваться, если у вас хоть что-то как-то получается, а краткость — не обязательно преимущество, она может быть и злом.

REPL

REPL — это аббревиатура от Read — Evaluate — Print Loop («цикл типа "прочитать — вычислить — записать"»). Вероятно, это звучит так, что вам хочется сказать: «О-о-о, это что-то слишком сложное, лучше просто пропущу». Но будет жаль, если вы так поступите, потому что REPL необычайно сильно помогает при изучении какого-либо языка.

REPL — в некоторых случаях его называют также интерактивной оболочкой (interactive shell) — принимает на вход коротенькие кусочки программы, выполняет их и выдает результат. Для языка программирования это то же, что оболочка (или — в случае с Windows — окно DOS) для операционной системы: интерактивная среда, в которую вы можете вводить команды, которые тут же выполняются, — ничего не надо компилировать или куда-то выгружать. Все вводимые вами команды заносятся в историю, так что позже вы снова можете к ним обратиться.

Так как в REPL вы видите непосредственный результат работы ваших команд и не тратите время на выгрузку и/или компиляцию, то они особенно хорошо подходят для программирования в пробных целях: вы играете с языком так и эдак, испытываете разные штуки. А когда делаете ошибки, видите их сразу.

Выражение «Read — Evaluate — Print Loop» появилось как обозначение интерактивной среды Lisp, так как этот язык с его пластичностью особенно хорошо подходит для работы по такому принципу. Однако такие интерактивные оболочки предусмотрены для очень многих языков. Возможно даже, что однажды вы уже работали с какой-нибудь из них.

- Реляционные базы данных, такие как MySQL, msSQL или PostgreSQL, уже десятилетиями снабжаются командными строками или GUI-клиентами, которые позволяют интерактивно составлять таблицы, а также осуществлять ввод и считывание данных. Вы вводите оператор `SELECT` и сразу получаете результат.
- Для языка Python существует довольно известный REPL iPython.
- В PHP есть оболочка phpsh (www.phpsh.org/).

- Вместе с интерпретатором Ruby поставляется REPL irb (Interactive RuBy).
- REPL для JavaScript встроен в большинство браузеров в виде консоли JavaScript. Стоит только открыть консоль, и вы тут же можете вводить команды JavaScript[1].
- Математически-статистические языки, такие как R, в значительной степени основываются на REPL, так как бóльшая часть процесса разработки ПО на таких языках — это моделирование алгоритмов анализа данных. Такое моделирование требует тестирования различных гипотез, что особенно просто и быстро делать с применением принципа пробной работы, используемого в REPL.

Для таких языков, как C++, исходный код на которых должен быть сначала полностью скомпилирован и которые используются скорее для системного программирования и создания графических пользовательских интерфейсов, REPL, напротив, встречается реже. Причина этого в том, что такие задачи плохо раскладываются на маленькие кусочки кода и их сложно решать интерактивно. И по крайней мере раньше на компиляцию программы уходило просто слишком много времени.

Пользуясь подобной интерактивной средой, вы можете объявлять переменные, которые будут сохранены в течение сеанса работы. Аналогичным образом вы можете определять функции, а позже обращаться к нескольким строкам. Это как калькулятор, только для языков программирования и круче.

Вот пример использования REPL irb для языка Ruby:

```
C:\scripts>irb
irb(main):001:0> puts "hello"
hello
=>nil
```

Здесь происходит следующее: в первой строчке мы с помощью команды `irb` запускаем Interactive RuBy, сокращенно irb. Потом вводим команду `puts "hello"`, которая есть не что иное, как команда печати. В тот момент, когда вы нажимаете на Enter, она выполняется и на экран выводится слово `hello`. Напоследок `irb` выводит выражение `=>nil`. Все, что следует за `=>`, — это возвращаемое значение команды Ruby. Хотя `puts` и выводит на экран текстовую строку, но в качестве функции не имела бы возвращаемого значения, поэтому здесь и появляется `nil`.

Совсем по-другому все выглядит в том случае, если вы складываете с помощью irb числа:

```
irb(main):002:0> 1+1
=> 2
```

Еще в REPL можно объявлять переменные, которые будут сохраняться в течение сеанса работы:

```
irb(main):003:0>pinkie_count = 3
=> 3
irb(main):004:0>pinkie_count -= 1
```

---

[1] Если вы не можете найти этот пункт меню, то используйте Firefox и установите расширение Firebug. В нем вы найдете консоль JavaScript под пунктом меню Консоль.

```
=> 2
irb(main):005:0> puts "Число мизинцев: #{pinkie_count}"
Число мизинцев: 2
=>nil
```

Здесь мы сначала определяем переменную `pinkie_count` и присваиваем ей значение 3. irb тут же его выводит. В следующей строчке исправляем значение переменной, вычитая 1. Выводится значение 2, что показывает, что переменная `pinkie_count` и дальше остается определенной и сохраняет свое значение. Ну а в последней строчке мы выводим ее значение с помощью команды печати (рис. 20.1).

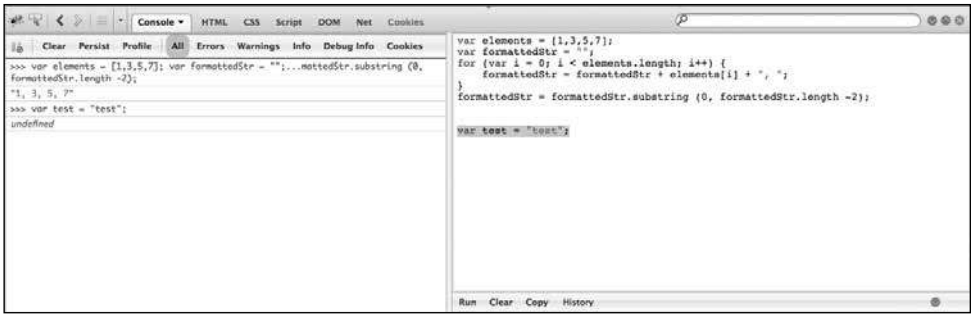


Рис. 20.1. Firebug в качестве REPL для JavaScript

Если у вас есть лишь приблизительное представление о том, как решить проблему, то REPL подойдет идеально: вы можете экспериментировать, пробовать различные идеи — и сразу будете видеть результат. Вам не придется сохранять файл и запускать его как программу. Эта интерактивность вызывает желание разрабатывать код небольшими кусками, будто это наброски в альбоме. Если вы уверены, что нашли верный путь, то можете скопировать код из истории и встроить его в свою программу. Если совсем запутались, можете все удалить и начать заново.

И для того, чтобы просто познакомиться с каким-нибудь языком и составить о нем впечатление, REPL — замечательная штука. Лишь немногие изучают язык, читая книгу или онлайн-документацию, — пока вы не решите с помощью языка пару небольших проблем, у вас не будет настоящего представления о том, насколько хорошо он вам подходит. Познакомиться с языком можно, только опробовав его, поиграв с ним.

Diff и patch

Если в вашем коде появилась новая ошибка, которую трудно найти, и вы используете тот или иной способ контроля версий или хотя бы время от времени выполняете резервное копирование, то вам поможет применение утилиты diff. Diff означает difference — «отличие». Это инструмент, который может сравнить два текстовых файла и отметить различия. Утилиты diff бывают как с графическим интерфейсом, так и для командной строки. О последнем варианте мы расскажем позже, когда будем говорить о patch-файлах.

Утилиты diff на базе графического интерфейса чаще всего имеют окно, разделенное вертикальной чертой на две части. На одной стороне показывается одна версия, на другой — другая. Слева от полей, отображающих код, находится полоса, показывающая различия в сжатом виде. С помощью вертикальной полосы прокрутки, находящейся с правой стороны, можно синхронно просматривать оба файла.

Простейшая версия утилиты diff есть в качестве дополнения во многих редакторах. Или же это может быть отдельная программа, например WinMerge или Diffmerge (рис. 20.2).

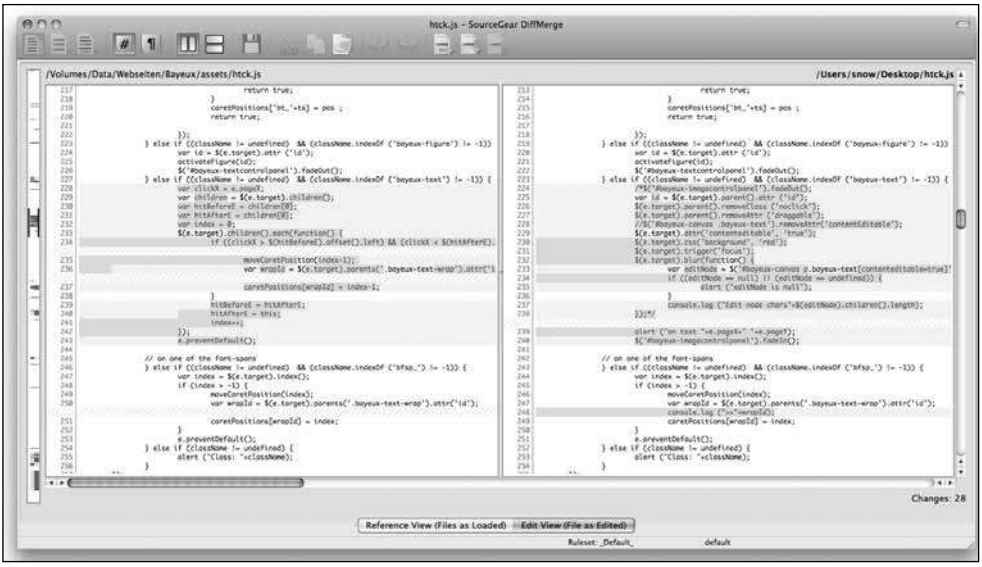


Рис. 20.2. Diffmerge в действии

Когда вы запускаете такую программу, она требует, чтобы вы открыли два файла для сравнения. Утилита diff способна сравнивать код в некоторой степени умно. В зависимости от выбранных программных установок она будет, например, полностью игнорировать различия в пробельных символах — это чертовски удобно в том случае, если с кодом работал более чем один программист и в результате табуляция была преобразована в пробелы. При «глупом» сравнении в такой ситуации оказался бы отмеченным весь текст, в то время как diff выделяет только блоки, различающиеся содержанием.

Часто в утилитах diff есть возможность выполнить объединение (merge). В данном случае под этим понимается то, что программа будет по нажатию кнопки копировать изменения из одной версии в другую, чтобы, например, сгладить конфликты в системе контроля версий. Только нужно очень четко решить, открывая оба файла, в какую сторону вы будете копировать, всегда справа налево или наоборот. И этого решения нужно очень строго придерживаться, потому что если вы скопируете что-то в неверном направлении, то уже не сможете ничего понять и лишь усилите хаос.

Если вы хотите поработать в не слишком тесном сотрудничестве с другими разработчиками — например, решили внести пару небольших изменений в проект с открытым исходным кодом, — то вам также стоит обратить внимание на программы diff и patch. Часто вам не будут сразу давать доступ к проекту с правом вносить изменения только для того, чтобы вы сделали пару улучшений, — вместо этого владельцы проекта просто попросят вас прислать patch-файлы.

Patch-файлы — это текстовые файлы, в которых построчно содержатся изменения, которыми различаются две версии какого-либо текста. Это выглядит, например, следующим образом:

```
145c149
< * Пользователи могут самостоятельно создавать новые аккаунты и просить
выслать себе забытые пароли.
---
> *Пользователи могут самостоятельно создавать новые аккаунты и восста-
навливать забытые пароли.
```

Цифры и буквы `145c149` в первой строке — это сообщение diff-программы, которое означает следующее: «В первой версии текста я нашел данное место в строке 145, во второй версии — в строке 149».

Символ `<` в следующей строке указывает, что эту строку нужно выкинуть, а символ `>` двумя строками ниже говорит: «Вместо нее надо вставить вот эту». Между этими patch-командами стоит еще символ `---`, который их разделяет.

Если между двумя версиями текста имеются различия более чем в одном месте, то в patch-файл с помощью такого синтаксиса записываются все изменения одно за другим.

Patch-файлы обрабатываются программой patch и создаются программой diff. С помощью обращения:

```
diffcode-version1.txtcode-version2.txt>code.patch
```

вы можете создать patch-файл с именем `code.patch`, который вы можете послать кому-нибудь другому. Тогда он может обратиться к patch:

```
patchcode-version1.txt<code.patch
```

и получит в результате текстовый файл, в точности соответствующий вашему.

У пересылки patch-файлов вместо целых файлов есть два преимущества.

- Получатель может заглянуть в patch-файл (это ведь простой текстовый формат) и быстро получить представление о том, что именно вы изменили.
- Patch проверяет состояние файла, который он должен обработать. Хотя это и не очень глубокая проверка, но если у получателя файл уже был изменен, то patch предупредит об этом.

Вот и все — внесенные вами изменения добавлены в файл получателя.

Тандем diff и patch может быть интересен не только программистам: например, это прекрасное средство для того, чтобы писать вместе книги, если вы не хотите месяцами пререкаться о том, какие инструменты и форматы подойдут лучше[1].

---

[1] Да-да. Или можно вместо всего этого просто использовать Документы Google.

Менеджер пакетов

Здесь повезло пользователям Linux, так как они с высокой вероятностью уже сталкивались с менеджером пакетов этой ОС при установке языков программирования или веб-серверов, — в зависимости от дистрибутива Linux это мог быть, например, dpkg/apt или rpm. Для Mac OS есть Mac Ports и Homebrew, правда, их нужно сначала установить. После этого для вас открыт весь мир свободного ПО — можете подбирать и устанавливать какие угодно программы. Пользователи Windows могут воспользоваться Cygwin или Npackd.

Конечно, свободное ПО, такое как языки программирования, серверы баз данных или веб-серверы, можно скомпилировать и самому, ведь исходный код легко скачать и это не запрещено. Однако это может стать занятием, за которым вы проведете весь вечер, потому что для компиляции большого пакета свободного ПО нужно много мелких инструментов командной строки и библиотек. Если в вашей системе стоит устаревшая версия какой-то библиотеки, то компилятор даст сбой и вам придется сначала установить новую версию, а потом уже продолжать установку нужной программы. А при компиляции библиотеки произойдет, конечно же, то же самое, потому что в вашей системе нет какой-нибудь другой нужной библиотеки.

В современном ПО очень многое зависит от библиотек, определенных версий компилятора или так называемых инструментов сборки (buildtools), то есть программ, которые помогают при компиляции. Распутывать вручную клубок этих зависимостей — неблагодарное занятие, и именно поэтому придумали менеджеры пакетов.

Менеджеры пакетов обеспечивают стандартизированный канал установки (свободного) ПО: они используют только ПО из своих каталогов (так называемых репозиториев), которое конфигурируется и структурируется по правилам пакетных менеджеров. Сопровождающие пакетов (это наши любезные современники, которые заботятся о том, чтобы с кодом мог работать определенный менеджер пакетов) делают за нас всю работу. Вы можете рассчитывать на то, что менеджер пакетов обеспечит надежное программное окружение — это гораздо легче, чем самому компилировать программы для всевозможных платформ во всех возможных версиях.

Еще одно преимущество менеджера пакетов в том, что он решает проблему зависимостей. Сопровождающий проекта должен лишь указать, какие библиотеки и инструменты сборки для какой версии он предусмотрел, и менеджер пакетов сам догрузит их при установке. Вас, как пользователя, может немного напугать то обстоятельство, что, когда вы хотите установить один только веб-сервер, менеджер пакетов устанавливает 50 других дополнительных пакетов — от языков программирования до SSL. Но это не должно вас отпугивать, ведь винчестеры нынче вместительны.

Если вы напрямую скачиваете и компилируете исходный код, то вам придется скачивать заново каждую новую версию, учитывать все зависимости, выяснять, не появились ли новые трудности при компиляции, и проверять, запускается ли новая версия на вашей системе. Все эти задачи может взять на себя менеджер пакетов,

и для того, чтобы установить актуальную версию нужной программы, вам нужно будет лишь набрать `apt-getupgrade` (или аналогичную команду для других менеджеров пакетов). Если вы по каким-то причинам (например, из-за несовместимости с другой программой, которую нельзя преодолеть с помощью менеджера пакетов) не хотите обновлять определенный пакет до самой новой версии, то это тоже не проблема: менеджер пакетов может и задерживать обновления.

В менеджерах пакетов часто есть функция поиска по репозиторию, с помощью которой вы можете целенаправленно искать пакеты. Если введете ключевое слово, например, `SSL`, то менеджер выдаст список пакетов — от SSL-сервера до SFTP-клиентов и веб-серверов с поддержкой SSL.

Основные функции большинства менеджеров пакетов похожи.

- `update` — менеджер пакетов загружает из сети список новейших версий пакетов.
- `upgrade` — эта команда устанавливает актуальную версию того или иного пакета. Как правило, есть и такой вариант команды `upgrade`, с помощью которой вы можете обновить все пакеты.
- `install` — устанавливает новый пакет. Для этого вам нужно ввести имя пакета.
- `remove` (или `delete`) — удаляет установленный пакет.
- `list` — выдает список недавно установленных пакетов, а часто и номера их версий.
- `find` — ищет какой-либо пакет в репозитории.

Наряду с «большими» менеджерами пакетов, такими как apt, во многих языках программирования есть и собственные менеджеры, специально ориентированные на разработчиков. В них вы не отыщете веб-серверов, зато найдете фреймворки и библиотеки, которые можете применить для собственного проекта и которые предложат вам решения, не предусмотренные в самом языке. Эти модули, часто имеющие название Package, содержат не только все файлы, составляющие библиотеку, но и метаданные: например, электронные адреса авторов, URL домашней страницы, а также информацию о тех самых зависимостях. Все эти метаданные считываются автоматически, поэтому процесс установки пакета, включая автоматическое скачивание и установку всех дополнительно необходимых пакетов, может быть автоматизирован.

Часто случается так, что для определенной проблемы, например обработки XML, существует несколько конкурирующих пакетов и создатели языка ни одному из них не отдают предпочтения. Тогда вы можете сами решить, какой вам наиболее симпатичен, и установить его с помощью встроенного в язык менеджера пакетов. Конечно, есть разные форматы пакетов и разные пакетные менеджеры, иначе все было бы слишком просто. Но не волнуйтесь: сообщества пользователей большинства языков договорились использовать для каждого языка только один формат.

Вот примеры менеджеров пакетов для некоторых известных языков:

- Perl — cpan;
- Ruby — gem;
- Python — pip;

- PHP — PEAR или Composer;
- Java — Maven2, Ivy или Gradle;
- JavaScript — Bower или Jam;
- Node.js — npm;
- .NET — NuGet;
- Objective-C — CocoaPods.

Иногда у вас есть возможность установить пакет как с помощью «большого» менеджера пакетов вашей ОС, так и с помощью менеджера, встроенного в язык. В этом случае мы советуем сделать выбор в пользу второго варианта, потому что зачастую это позволяет установить более актуальные пакеты. Рано или поздно они будут включены и в репозиторий пакетного менеджера вашей системы, но нет особых причин этого ждать. Кстати, следует по возможности избегать установки одного пакета с помощью обоих менеджеров, потому что тогда вы быстро потеряете контроль над тем, какая версия у вашего ПО.

Фреймворки

Если вы хотите разработать программу более сложную, чем какой-нибудь маленький сценарий командной оболочки, то вам не обойтись без использования чужого кода, который обеспечит выполнение базовых функций, таких как доступ к базе данных или парсинг XML. Также готовый код часто используют при создании графического пользовательского интерфейса или веб-приложений.

Полезный код, который реализует общие, вновь и вновь используемые базовые функции, обычно поставляется в форме пакетов, называемых библиотеками (см. главу 19) или фреймворками. Провести границу между этими двумя понятиями не всегда просто, в частности, потому, что иногда они четко не разделяются.

В качестве грубого правила можно принять следующее: библиотека предоставляет функции, к которым можно обратиться из собственной программы. Как писать программу и как использовать библиотеку — решать самому разработчику. В противоположность этому фреймворк как тоталитарное государство: если не хочешь жить по его порядкам, остается только эмигрировать и начать использовать другой фреймворк. Потому что фреймворки задают довольно строгие правила, согласно которым приходится структурировать программу.

Чуть более по-научному: библиотека предоставляет решения для какой-либо определенной проблемной области (например, XML-парсеры упрощают считывание XML), в то время как фреймворки представляют собой каркас для всей программы, в определенные места которого вы вставляете свой код согласно правилам фреймворка. Примером могут послужить фреймворки веб-приложений, такие как Ruby on Rails, которые облегчают быструю разработку динамических веб-сайтов. Библиотеки вы подключаете к какой-либо программе и обращаетесь к их функциям, в то время как фреймворки работают по принципу Голливуда: «Не звоните нам, мы позвоним вам сами». Это значит, что программист больше не пишет метод `main` и не запускает программу, а вместо этого определяет в файлах конфигурации фреймворка, какие функциональные возможности он хочет

получить, задает шаблоны HTML или кода, определяет схемы баз данных, а потом пишет только тот код, который является специфическим для его задачи.

Это ничуть не хуже, чем использование библиотеки. Пускай все и звучит пугающе, но использование фреймворков имеет ряд преимуществ.

- У не слишком опытного программиста, использующего фреймворк, меньше возможностей для ошибки. Если фреймворк и неидеален, его философия по крайней мере не окажется полностью непригодной.
- Хорошие фреймворки выстраивают логичный простой каркас, использование которого избавляет разработчика от множества скучной работы.
- Опытные разработчики знакомы с философией, согласно которой был спроектирован фреймворк, и могут соответствующим образом подстраивать под нее свой код.
- Не изобретать велосипед заново — это всегда преимущество в деле разработки ПО.

Выбор фреймворка

Когда вы выбираете фреймворк, вы не то чтобы продаете за него душу, но по крайней мере на срок проекта на нем женитесь. И от этого выбор идеального партнера еще тяжелее. Можно выбрать фреймворк, который берет на себя слишком много задач и именно из-за многофункциональности перестает быть понятным. А можно выбрать фреймворк, функций которого окажется недостаточно. Если вы сомневаетесь, то лучше выбрать слишком скромный фреймворк: тогда легче понять, как с ним работать. Отсутствующие функции вы либо реализуете сами, либо надеетесь на новую версию, где их количество будет расширено. Слишком сложный фреймворк приводит скорее к тому, что проектом перестают заниматься и обрекают его на смерть.

При работе с фреймворком может настать момент, когда вы подумаете: «Да какого черта им надо, без фреймворка я решил бы эту проблему в пяти строчках!» Это нормально. Фреймворки рассчитаны на самые разные случаи применения, и поэтому они настолько обобщенные, что иногда простое в них становится сложным. Однако альтернатива, заключающаяся в том, чтобы самому все написать с нуля, кажется более простой, только пока программа маленькая. Когда она становится большой и сложной, каждое изменение превращается в весьма рискованное предприятие. Поэтому, когда вы решаете, использовать фреймворк или нет, стоит прикинуть, что потребует большего количества работы, самому реализовать необходимые базовые функции или понять выбранный фреймворк и написать код и файлы конфигурации, чтобы с ним можно было работать.

Предупредительные знаки при работе с фреймворками

Если у вас постоянно возникает чувство, что фреймворк лишь вставляет вам палки в колеса и все гораздо сложнее, чем должно было быть, довольно велика вероятность того, что вы не поняли основополагающую концепцию фреймворков и пы-

таетесь писать ваш код в рамках какой-то другой концепции. Некоторые называют это борьбой с фреймворком, некоторые — вколачиванием квадратных кольев в круглые дырки.

В целом фреймворки следуют некоторой философии, определяющей, каким образом должна быть организована слаженная работа библиотек, шаблонов, ресурсов и исходного кода, являющегося для проекта специфическим. Эта философия оказывает влияние и на ваш код, потому что только если вы будете следовать концепции фреймворка, вам удастся писать программу без особых проблем. В противном случае у вас всегда будут вызывать удивление моменты вроде следующих.

- «Мне надо прямо здесь и сейчас получить от фреймворка соединение с базой данных, но я не нахожу никаких способов это сделать». Вероятно, причина проблемы в том, что фреймворк хочет удержать разработчика от реализации прямого доступа к базе данных, а вместо этого предлагает ему интерфейс, с помощью которого можно будет избирательно записывать данные в БД, файлы или на удаленный сервер. Вместо того чтобы искать соединение с базой данных, следует выяснить, каким образом вы можете передать объекты фреймворку для хранения.
- «Мне надо получить на этом месте результаты, которыми я хочу еще раз воспользоваться позже, но они, увы, еще раньше были переданы в базу данных или пользователю». Здесь проблема может быть в том, что фреймворк предусматривает многоступенчатую обработку, при которой сначала проверяются параметры, затем получаются данные и, наконец, выдаются значения. Возможно, ваш код расположен на неправильной ступени.

Новичкам сложно отличить чувство, что ты работаешь против фреймворка. И без того приходится постоянно барахтаться изо всех сил, чтобы оставаться на плаву. Но есть случаи, в которых хочется просто заорать: «НЕТ! ЭТО НЕ МОЖЕТ БЫТЬ НАСТОЛЬКО СЛОЖНЫМ!» Тогда стоит задуматься: а не может ли оказаться так, что дело не в фреймворке, а в том, что вы пытаетесь использовать отвертку в качестве молотка?

Если в названии фреймворка встречается слово enterprise, это верный предупредительный знак. Речь идет не о забавном космическом корабле, а о бюрократическом монстре, который был создан для использования в крупных концернах. Enterprise-фреймворки пишутся не для того, чтобы облегчить работу в какой-либо четко очерченной тематической области, — они пишутся с оглядкой на список возможностей. Неважно, каких именно, — это могут быть аспектно-ориентированное программирование, облачные технологии, Map/Reduce или любые другие разрекламированные возможности. Разрекламированные, к сожалению, чаще всего незаслуженно. Причина этого в том, что в крупных концернах решения принимают не разработчики, а их начальники двумя уровнями иерархии выше — и они, с одной стороны, боятся риска, а с другой — редко бывают знакомы с актуальными трендами. Поэтому в случае сомнений они с большей охотой выбирают фреймворк с впечатляющим списком возможностей, разработанный крупной фирмой, который в 95 % случаев избыточен для предполагаемой области применения.

Среды разработки

Среды разработки среди инструментов для разработки ПО будто швейцарские армейские ножи. Их идея в том, чтобы совместить в одной оболочке как можно больше инструментов, нужных разработчику, отсюда и происходит их английское название Integrated Development Environment (IDE) — интегрированная среда разработки. Они включают текстовый редактор, систему управления проектом и, как правило, другие средства, такие как система контроля версий и инструменты для рефакторинга. IDE особенно хороши тем, что включают в себя программу-отладчик, которой очень удобно пользоваться: можно написать программу и тут же одним нажатием кнопки запустить в отладчике, чтобы проверить ее работу.

Такое объединение невероятно облегчает жизнь программиста. По крайней мере тогда, когда вы привыкли работать в какой-либо среде разработки. А до тех пор на освоение придется потратить немало времени и нервов. Потому что у всех сред разработки есть общая черта: с ними не научишься работать за один вечер.

Лучше всего это заметно на примере очень широко распространенной среды разработки с открытым кодом Eclipse. Она была задумана как панацея от всех бед в сфере разработки ПО и поэтому представляет собой лишь каркас для IDE, который можно расширять и конфигурировать в любом направлении. По сути, ее функциональность достигается с помощью плагинов, то есть расширений. Можно скачать те из них, которые поддерживают используемые вами языки программирования или фреймворки, создав для себя что-то вроде среды разработки по индивидуальному заказу. Поэтому Eclipse может применяться для разработки практически на любых языках программирования, а кроме того, использоваться как редактор HTML.

Управление проектами

Многие среды разработки организуют исходный код в проекты. Это позволяет работать с кодом для разных программ в одной и той же IDE. Файлы не смешиваются, так как каждый помещен в определенный отведенный для проекта ящичек — один-единственный. Даже если вы создадите в разных проектах файлы с одинаковыми именами, среда разработки будет знать, какому проекту принадлежит какой файл. На первый взгляд это не кажется чем-то удивительным — каждый проект является каталогом в файловой системе, то есть среда разработки упорядочивает файлы проекта точно так же, как вы бы сделали это сами.

Кроме того, среда разработки подшивает к каждому проекту информацию о том, что требуется для превращения кода в рабочую программу. Различия от языка к языку здесь очень велики: в случае программы на C++ это может быть компилятор или запуск компоновщика, в случае приложения Java Enterprise должен быть создан JAR-архив или запущен веб-сервер. Есть также среды разработки для веб-приложений, которые или запускают локальный веб-сервер, или могут загружать файлы на сервер через SSH. А в случае мобильных приложений к компиляции добавляются нанесение электронной подписи и выгрузка в эмулятор или на смартфон. Все это указывается отдельно для каждого проекта и собирается в Build Configuration (конфигурации

сборки). Если вы хотите экспортировать готовую версию программы, то нажимаете кнопку Build, и в недрах среды разработки тут же запускается машина, которая проделывает все шаги, остающиеся до завершения продукта.

Среды разработки могут также управлять зависимостями программ, которые вы пишете с их помощью. Это значит, что вы указываете библиотеки или фреймворки, которые хотите использовать для создания программы, и среда разработки может добавить соответствующие файлы к нужной конфигурации сборки. Еще среды разработки часто могут включать в себя менеджеры пакетов (см. раздел «Менеджер пакетов» ранее в этой главе), такие как Maven или Bower: вы прямо из IDE ищете нужную библиотеку в списке и определяете нужную версию, а IDE скачивает требуемый файл. Это довольно удобно, если хотите запустить более новую версию библиотеки в тестовом режиме: вы можете прямо в IDE изменить номер версии на нужный, она скачает новую версию и скомпилирует вашу программу, и вы сможете проверить, не приводит ли смена версии к ошибкам при компиляции или исполнении программы.

Проверка кода

Ежедневный трудовой процесс профессионального программиста чаще всего выглядит так: извлечение кода из системы контроля версий, программирование, проведение рефакторинга, тестирование, занесение изменений в систему контроля версий. Именно это облегчают IDE. Сразу хотим успокоить тех, кому при словах «профессиональный программист» захотелось отложить книгу в сторону: инструменты, которые предлагают IDE, полезны для всех, даже для новичков, которые еще вчера ломали голову над тем, как вывести на экран фразу «Hello, World!».

Среды разработки включают в себя парсер языка с проверкой синтаксиса, который еще во время написания проверяет код на соответствие правилам синтаксиса (рис. 20.3). Этот редактор понимает синтаксис определенного языка и отмечает код с ошибками. Это экономит время, так как вам не приходится сначала компилировать файл с кодом, содержащим ошибки, или загружать его на сервер, для того чтобы эти ошибки обнаружить. (Справедливости ради стоит заметить, что все неплохие текстовые редакторы, такие как emacs, vim, Sublime и т. д., тоже это умеют.)

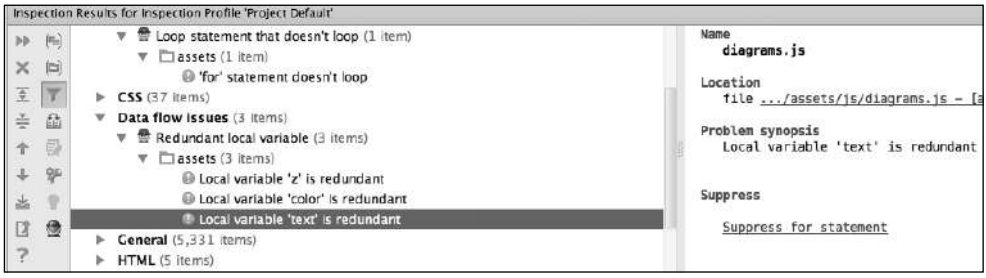


Рис. 20.3. «Технадзор» за кодом в действии

В языках со статической типизацией (см. главу 26), таких как Java, IDE также проверяют, не хотите ли вы передать функции, которая ожидает число на входе, файл, строку или еще какую-нибудь переменную неправильного типа. На рис. 20.4 Eclipse любезно подсказывает, что Java не поддерживает автоматическое преобразование строкового типа в числовой, поэтому вы не можете умножить строковую переменную на число.

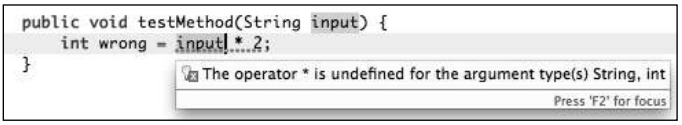


Рис. 20.4. Java не любит умножать строки

Среды разработки могут отмечать код, который хотя и не содержит ошибок, но вызывает так называемые предупреждения (warnings). Отмеченный код затем отправляется в список проблем или предупреждений. Небольшие проблемы с кодом, такие как пропуск точки с запятой в конце строки или избыточные проверки на `NULL`, возникают всегда, и в спешке им можно не придать значения, однако у них есть свойство перерастать в настоящие ошибки при неблагоприятном стечении обстоятельств. Среда программирования, прививающая пользователю правила хорошего тона, помогает избежать огорчений в будущем.

Автодополнение кода

В то время как текстовый редактор можно просто открыть и сразу начать программировать, в средах разработки вам придется организовать свою работу по проектам. В этом случае среда разработки знает наверняка, какой язык или языки, какие библиотеки и какие фреймворки вы собираетесь использовать для каждого проекта. Основываясь на этом знании, она может — по вашему желанию — делать предложения по автодополнению кода (рис. 20.5).

Рис. 20.5. Автодополнение кода в Eclipse

В этом примере показан кусок программы на Java. `Selection` — это локальная переменная строкового типа. IDE знает, какие методы применяются в Java к стро-

кам, и при вводе этой переменной предлагает их в качестве вариантов автодополнения. Так как Java — язык с сильной типизацией, IDE может также предоставить данные о том, сколько параметров какого типа ожидает каждый метод и каково возвращаемое значение того или иного метода (рис. 20.6).

Рис. 20.6. Скоропись в Eclipse

Преимущества сред разработки

Наверное, для меня это та возможность, за которую я на самом деле люблю среды разработки: ввести имя переменной, после нее — точку и смотреть, что можно делать с этим объектом. Если я вижу, что ничего не появляется, то я знаю, что где-то в программе допустил ошибку и должен сперва позаботиться о ее исправлении. Это краткая справка, отладчик и ускоритель работы, соединенные вместе. И к тому же постоянное повышение квалификации: «О, смотри-ка, этот объект может еще и это!»

Эта возможность позволяет мне чувствовать себя уверенно даже с такими ужасно недружелюбными продуктами, как Oracle SQL Developer — IDE для программирования баз данных. Когда работаешь в довольно гетерогенной среде со многими разными стандартами присвоения имен графам таблицы, то очень удобно просто ввести «USER.» и увидеть, как называется графа первичного ключа: ID или USER_ID. Кроме того, каждый тип, каждая переменная и каждая функция представляют собой гиперссылку, так что можно очень легко перейти к их определению.

Йоханнес

Сходна с этим и система, позволяющая вводить только краткую форму типов переменных[1]. По сокращению IDE может отгадать и отобразить тип, который вы ищете: `CDSE` превращается в `ConfigurableDSEMessage`. Это особенно удобно для тех языков, в которых много длинных имен типов, записываемых в верблюжьем регистре. Вы набираете пару прописных букв из начала или середины, а IDE вытягивает их в полное имя типа. В языках, где типы не задаются и у всех переменных тип `var`, эта функция, конечно, не столь полезна.

---

[1] Следующий пример взят из объектно-ориентированного языка. В таких языках можно самому определять свои типы переменных в форме объектов, как в примерах из главы 23 с типами Article, PngImage или Customer.

Помощь в организации труда

Многие среды разработки предлагают возможности, помогающие программисту организовать его труд. Сюда относится наличие системы контроля версий (Version Control System, VCS). Конечно, без проблем можно после каждого значительного изменения сворачивать текстовый редактор и лезть на Рабочий стол или в терминал, чтобы там внести последние изменения в систему контроля версий. Но когда вы отрываетесь от кода, потом можете обнаружить себя на reddit — проверяющим, как развивается ситуация с мемами про котиков, — и вот вы уже сбились с рабочего ритма (рис. 20.7).

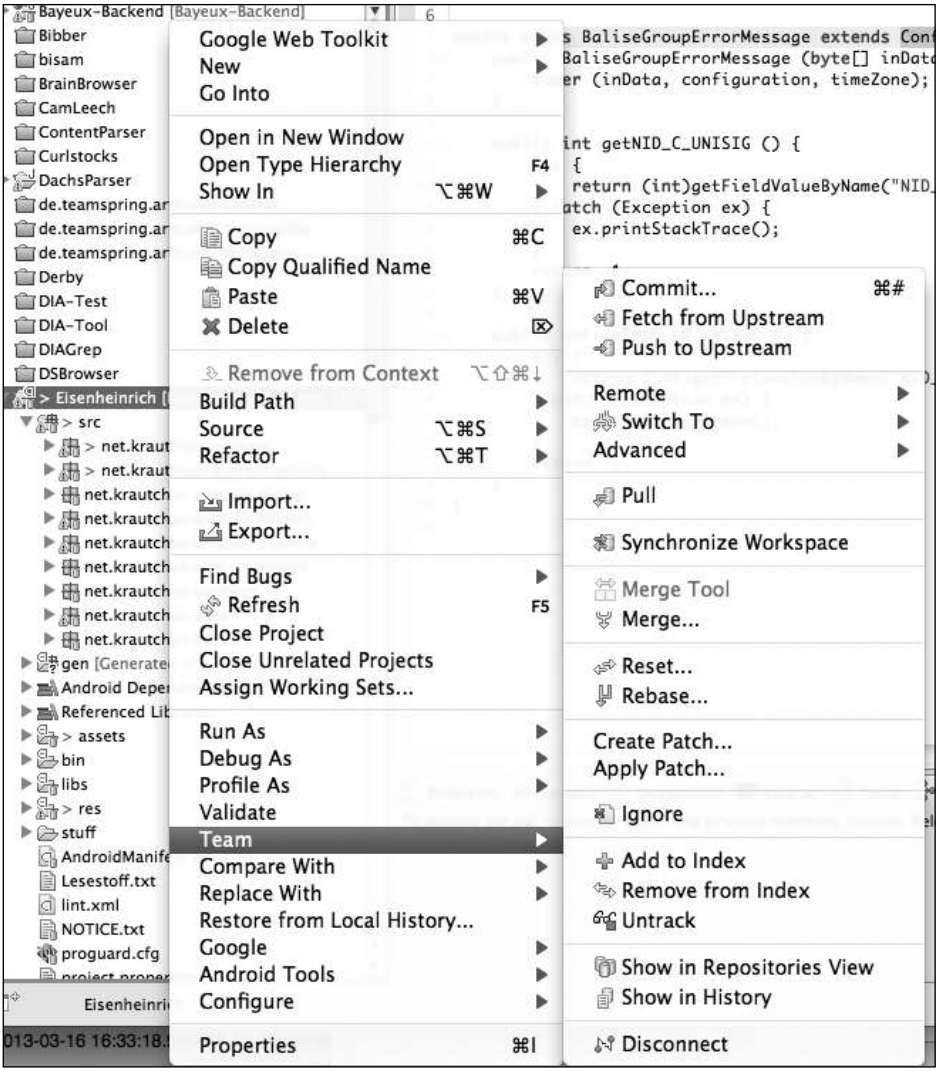


Рис. 20.7. Подключение Git Eclipse

В среде разработки вы можете выбрать и занести в систему контроля версий целый проект, подпроект или отдельный файл, а также вернуться к исходному состоянию VCS или просмотреть список изменений. Здесь нет ничего, что нельзя было бы сделать и без IDE, но зато все под рукой.

Запланированные действия можно записывать в комментариях — в разных языках по-разному, например, так:

```
// TODOErrorhandlingomittedfornow (ошибка пока что не устранена)
```

И по нажатию кнопки IDE покажет вам непочатый список дел. Если у вас, например, набит живот после обеда и вы сейчас не способны на настоящую креативность, то можете посвятить себя работе с этим списком. Есть похожие списки и для ошибок и предупреждений.

В средах разработки имена переменных и типов часто являются гиперссылками. Можно щелкнуть на имени переменной, и текстовый редактор тут же перейдет к ее определению. С типами переменных еще удобнее: когда вы на них щелкаете, открывается файл, в котором они определены. Это преимущество важно также только для объектно-ориентированных языков с сильной типизацией, так как лишь в них во время программирования понятно, какой тип имеет та или иная переменная. В слабо типизированных и динамических языках, таких как PHP или Perl, IDE не может этого определить.

Поиск и рефакторинг

Среды разработки — мощные инструменты программирования и сопровождения кода. Тот, кому доводилось погубить проект в результате глобального поиска и замены имени переменной, ценит тот факт, что IDE понимает код. С ее помощью можно изменить имя переменной целенаправленно, не меняя заодно и одинаково звучащие имена функций и значения констант. Представьте, что в следующем отрывке кода вы хотите заменить переменную `username` на `user_id`.

```
const prompt = "Please enter your username" ("Пожалуйста, введите ваш
логин")
varusername = get_username();
```

При этом вы, вероятно, не хотите менять строку, которая применяется в пользовательском интерфейсе, менять имя функции вам, возможно, тоже не нужно. Вы хотите изменить лишь имя переменной. Решить эту задачу с помощью поиска и замены было бы трудно, а вот среда разработки понимает разницу между переменной, именем функции и строковой константой, и она может вам очень сильно помочь. Интегрированный поиск часто может различать поиск в именах функций и поиск в комментариях — это делает его результаты значительно более ценными.

Если файл стал слишком длинным, то его отдельные составляющие, такие как переменные, функции или внутренние классы, придется кропотливо копировать, распределяя по новым файлам. Потом необходимо удостовериться, что новые модули импортируются так же, как прежний, с которым стало неудобно работать. А потом вы еще должны проверить, какие дополнительные изменения нужно внести в код. В среде разработки вы можете выделить часть кода или внутренний класс

и автоматически переместить их в новый файл. Все взаимосвязи IDE автоматически перетащит следом. С такой помощью разбивка слишком сильно разросшихся файлов будет вселять гораздо меньший ужас.

Недостатки

Среды разработки никогда не побеждают в конкурсах на самое высокое юзабилити, и, по правде говоря, их интерфейс всегда перегружен: кнопки для важных команд, такие как Скомпилируй и запусти программу!, расположены рядом с кнопками, которые нужны гораздо реже, вроде Новый С-файл. Эти черты — недостаточно дружественный интерфейс и ужасно большое количество функций — могут отпугнуть неопытных пользователей. Приятным исключением среди традиционных сред разработки являются продукты майкрософтовской Visual-Studio-Express. Eclipce — флагман среди свободного ПО — к сожалению, являет собой хороший пример плохого дизайна оболочки. Среды разработки для различных языков создает фирма JetBrains, и хотя ее продукты безобразны, они довольно хорошо продуманы и неплохо помогают.

Другие недостатки таковы.

- Среду разработки придется вначале найти и установить. Это скучно и порой трудно. Поиск по запросу «IDE+ название языка программирования» должен выдать несколько ссылок, между которыми потом можно выбирать.
- Затем нужно освоить работу со средой. Количество функций велико, а интерфейс скорее плох, поэтому кривая обучения крутая. Для зеленых новичков это минус, так как им приходится одновременно справляться и с проблемами в новом языке, и с освоением IDE.
- Дурные привычки лишь в очень незначительной мере распознаются и исправляются средой разработки. Вы без проблем можете писать ерундовый код и в IDE.
- Хотя вы и можете выбирать из различных IDE, но, когда выбор сделан, вы связаны особенностями конкретной среды. Среда разработки объединяет различные инструменты, и работать с ней будет лучше всего в том случае, если предусмотренного в среде ящика инструментов вам хватает. Тот, кому в ИКЕА не хватает пяти стандартных цветов серии «Билли», будет, наверное, более счастлив без IDE. С помощью терминала и всего прочего он напрямую приспособит рабочую среду к своим потребностям, и ему не нужно будет идти на компромиссы. Тем же, кто предпочитает плохой компромисс длительной конфигурации, стоит хотя бы попробовать поработать в IDE.

Возможно, вы испытываете смутное нежелание как-либо менять ваш нынешний инструментарий. Жили же как-то и без этого! Против этого нечего возразить. Тогда можете смотреть на эту главу как на чужую мастерскую. Теперь вы хотя бы узнали о существовании описанных здесь инструментов и о некоторых возможностях их применения. Возможно, в один прекрасный день наступит момент, когда вы почувствуете, что что-нибудь из этого смогло бы помочь вам решить вставшую перед вами проблему.

Ситуация не сильно различается для вас и опытных программистов: неважно, как много вы знаете, — всегда найдутся инструменты и стратегии, о которых вы даже не слышали. Разве что у более опытных со временем появляется чутье, благодаря которому они понимают, при решении проблем какого рода можно надеяться на существование готового решения или инструмента, облегчающего работу. У них есть стратегии, позволяющие отыскивать эти решения и постоянно мимоходом следить за тем, не появилось ли что-то новое. И когда речь заходит об изменении процесса их работы, они порой упираются меньше, чем неопытные программисты, если это изменение сулит преимущества.

21 Система контроля версий

> Team Fortress 2 установлена на рабочий компьютер. Если возникнут вопросы, это не программное обеспечение контроля версий и обзора кода.
>
> *Tim Weber/@scy, Twitter, 17 января 2011 года*

Система контроля версий (VCS) — это напоминалка и спасательный якорь для программистов. Данная система дает возможность в любое время сохранить текущее состояние программы, а затем при необходимости вернуться к любому моменту. Система функционирует по принципу сохранения очков в игре: если игрок чувствует уверенность в середине игры, он сохраняет текущее состояние (сохраняется). И если позднее он погибает от нападения зубастого монстра, игра продолжается с сохраненного состояния.

Система контроля версий позволяет разработчику работать гораздо свободнее, без страха. Если в программе была допущена ошибка, которую трудно отладить, вы можете либо усердно просматривать программу до тех пор, пока не найдете ошибку, либо, вернувшись к старой версии, создать новую (и, вероятно, лучшую).

Для разработчиков программного обеспечения система контроля версий является также специализированной системой резервного копирования данных. Однако резервные копии создаются не просто через определенный интервал. Разработчик создает резервную копию (он фиксирует или регистрирует код), когда уверен в том, что стоит прекратить работу над текущей версией. Этот принцип известен всем, кто хоть раз редактировал статью в «Википедии»: для всех статей ведется история изменений, благодаря которой можно ознакомиться с различиями между двумя версиями одной статьи. При необходимости администраторы «Википедии» могут восстановить более раннюю версию статьи. Если статья была изменена, ее сохраняют как новую версию. Разница в использовании VCS в том, что вики-авторы зачастую изменяют отдельную запись, а разработчики ПО в процессе программирования, как правило, обрабатывают новые возможности в большом количестве файлов и хотят синхронно сохранить все эти изменения как новую версию. Таким образом будет упрощен так называемый откат к предыдущей версии, так как одновременно будут отменены все взаимозависимые изменения в используемых файлах.

В то время как программы резервного копирования отображают только время резервного копирования и восстанавливают содержание, VCS позволяет опреде-

лять, кто и когда сделал то или иное изменение. К тому же, как правило, VCS для каждой фиксации запрашивает комментарий, в котором кратко описано, что было реализовано. При определенных обстоятельствах эта дополнительная прозрачность позволяет быстрее обнаруживать новые ошибки, так как необходимо проанализировать только измененные строки кода.

Большая часть обычных VCS — бесплатное программное обеспечение с открытым исходным кодом. Вы можете пользоваться своей VCS даже со старого ПК или арендного сервера. Тем не менее вам все равно необходимо создавать резервные копии и администрировать сервер, потому что, если сервер, установленный в подсобке у вас в офисе, выйдет из строя из-за пожара или взлома, очень вероятно, что ваш рабочий компьютер, VCS и резервные копии также выйдут из строя.

В таком случае стоит подумать о решении проблемы с помощью хостинга. Если программное обеспечение с открытым исходным кодом не является хорошей платформой для собственного VCS-сервера, вам следует, исходя из лучшей обозримости своего проекта, выбрать самый большой хостинг-провайдер. В настоящее время наиболее известными являются следующие:

- sourceforge.net. Один из старейших хостинг-провайдеров для проектов с открытым исходным кодом, предлагающий различные системы контроля версий (CVN, Git и Mercurial). Кроме того, Sourceforge поддерживает историю загрузок и дискуссионные форумы;
- github.com. Несмотря на то что github произвел переворот в мире VCS, работающих на хостинге, остальные хостинг-провайдеры сокращают дистанцию. Для поддержки иностранных проектов предлагаются бесплатные хостинги проектов с открытым исходным кодом, хостинги приватных репозиториев на платной основе, веб-редакторы для различных файлов и другие удобные возможности. Github удалось добиться такого успеха в связи с применением компонента Social Coding, благодаря которому в совместной работе над проектами с открытым исходным кодом были впервые введены элементы социальной сети (наряду с размещением кода участники могут общаться, комментировать правки друг друга, а также следить за новостями знакомых). Github основан на системе контроля версий Git;
- bitbucket.com. Bitbucket так же, как и github, предлагает хостинг небольших приватных репозиториев, но на бесплатной основе. В качестве системы контроля версий используются Git и Mercurial;
- code.google.com. Несмотря на оптическую и техническую отсталость, не теряет популярности у разработчиков, в частности у тех, которые участвуют в таких конкурсах, как Google Summer of Code. Доступны системы контроля версий Mercurial, Git и CVN-Hosting, а также вики-источники по конкретным проектам и инструменты обзора кода. Code.google.com хранит только проекты с открытым исходным кодом.

Если вы хотите разработать ПО, код которого не хотите открывать в ближайшее время, то вам придется заплатить за VCS-хостинг.

Все основные VCS-хостинги предлагают вместе с хостингом проектов также систему отслеживания ошибок.

Альтернативы

Многие начинающие программисты и программисты, работающие нерегулярно, противятся тому, что, кроме работы с абсолютно необходимым инструментарием, их втягивают в работу в VCS. Если вы принадлежите к группе таких программистов, не стоит закрывать глаза на эту проблему, а следует как минимум найти альтернативное решение. Однако такой вариант возможен только в том случае, если вы программируете самостоятельно. При работе в команде VCS является необходимостью, потому что только таким образом код сможет остаться согласованным. Самый ужасный кошмар в командной разработке — это когда два члена команды реализуют различные возможности и при этом вносят изменения в одинаковые файлы. Если у них нет VCS, то один переписывает изменения, внесенные другим, и программа становится нерабочей либо в ней не хватает расширений, написанных другими членами команды.

Но поскольку вы читаете эту книгу, вряд ли вы член команды разработчиков. Если не хотите использовать систему контроля версий, вероятно, вы просто неряшливы, поэтому хотя бы приберитесь в ванной и попробуйте найти инструмент получше.

Преимущества VCS настолько велики, что компания Apple даже встроила в операционную систему Mac OS[1] программу Versions специально для контроля версий. Когда вы изменяете файлы, операционная система в фоновом режиме создает архивированные версии. Если вы хотите зафиксировать определенный момент состояния, можно указать версию вручную. Операционная система позволяет перебирать версии для возврата к предыдущей. Versions можно использовать для разработки ПО, но только при условии, что реализация возможности требует изменения лишь некоторых файлов. Для полноценной системы контроля версий это не проблема, вы фиксируете взаимозависимые изменения во всех файлах, Versions же, наоборот, работает на уровне отдельных файлов, и если вам не повезло, у вас не окажется одной версии одного определенного файла. Пользователи Mac-OS могут применять Versions для программирования небольших проектов, исходники которых состоят из малого количества файлов.

Но самая плохая альтернатива — это создание резервных копий. Внешний диск и программа резервного копирования дают возможность в любое время вернуться к более ранней версии кода. Резервные копии — это то, что вы в любом случае делаете, и хотя они не могут заменить все возможности VCS, они дают уверенность в том, что код не пропадет, если у вас украдут ноутбук. Если однажды вам понадобится развернуть резервную копию, вы вернетесь к тому состоянию, когда эта копия создавалась, но оно может быть совершенно отличным от того, которое вы хотели бы восстановить. Кроме того, нет никаких комментариев, относящихся к точке фиксации (коммит-комментариев), поэтому вы не знаете, к какой версии вернетесь. Резервное копирование подходит только для программиста, работающего в одиночку, так как оно не позволяет синхронизировать исходный код, созданный двумя разработчиками. Еще один недостаток резервных копий в том, что для реализации возможности зачастую требуются изменения некоторых

---

[1] С 2011 года Mac OS X Lion.

файлов. Если резервная копия сделана в то время, когда вы реализовали только первую часть изменений, постарайтесь развернуть ту версию кода, которая соответствует нерабочей программе.

Если вам до сих пор не удалось приучить себя регулярно — в данном случае не имеется в виду ежегодно — делать какие-либо резервные копии, то не тешьте себя иллюзиями, что это вдруг получится в будущем лишь по той причине, что вы намереваетесь однажды все-таки начать жить правильно. Лучше поискать какое-нибудь техническое решение, которое поможет решить проблему с мотивацией. Между тем существуют такие провайдеры платных автоматизированных облачных решений для резервного копирования, как Dropbox, iCloud и Backblaze, которые без всяких размышлений и принуждений через определенное время сохранят важнейшие данные с сервера на другом конце света, а по желанию даже зашифруют.

Ели вы считаете облако недостаточно приватным, то можете с помощью задания cron (см. главу 22) как минимум раз в день собирать важнейшие каталоги с сервера в ZIP-архивы. Профессионалы утверждают, что три резервные копии — две локальные и одна вне дома — дают шанс в случае чрезвычайной ситуации сохранить хотя бы одну рабочую версию.

Решение с применением cron может выглядеть приблизительно так: инструмент командной строки rsync (стандартный для Mac OS X и Linux, может быть переустановлен на Windows) позволяет вам регулярно, например каждый час, копировать содержание разрабатываемых каталогов с сервера. Даже если что-нибудь случится, по крайней мере вчерашнее состояние может быть восстановлено. Запись в Crontab должна была выглядеть примерно так:

```
* * * * * rsync -arze ssh --delete ~/development/ 192.168.1.20:/backups/
development
```

В частности, это означает: программа rsync должна передать по SSH, то есть по зашифрованному соединению, содержимое папки `development` из вашего домашнего каталога на компьютер с IP-адресом 192.168.1.20 и там сохранить эту информацию в каталоге `/backups/development`. Если вы локально (у себя) удаляете файл, то и на сервере резервного копирования он будет удален.

Работа в VCS

Работа в VCS вне зависимости от системы выполняется сходным образом: в ходе работы над проектом первое, что необходимо сделать, — извлечь актуальное состояние проекта из VCS (Check-out). Это следует делать ежедневно или после каждого сохранения, чтобы как можно раньше обнаружить возможные конфликты и устранить их. Только после успешного извлечения можно работать с тем же состоянием, что и все остальные.

Каждый сохраняет свои изменения локально, и хотя VCS замечает эти изменения, она никак не реагирует.

Когда достигнуто стабильное состояние, изменения загружаются в репозиторий VCS (Commit или Check-in). Репозиторий VCS — это место хранения всех версий

всех файлов всех проектов, которые обслуживаются системой контроля версий. Выполняя сохранение, настоятельно рекомендуется дать краткое описание того, что было сделано. Это коммит-комментарий.

Если были созданы новые файлы, они добавляются в VCS (Add). После каждого добавления необходимо выполнить сохранение, чтобы загрузить файлы в репозиторий. Здесь новичков подстерегают две ловушки: либо разработчик забывает добавить новые файлы и его коллеги их не получают, либо после добавления он забывает выполнить сохранение, из-за чего файлы идентифицируются как загрузки и коллеги все равно их не получают. Система контроля версий настаивает на добавлении, управляемом вручную, чтобы в рабочем каталоге могли храниться файлы, которые должны остаться на локальном уровне.

В некоторых VCS, например Git, существует такой отдельный шаг, как команда Push, когда версия сначала сохраняется в локальный репозиторий. С помощью команды Push она будет загружена и опубликована. Как именно это функционирует, будет разъяснено позднее в описании Git.

Если кто-нибудь одновременно с вами изменяет один и тот же файл, необходимо слияние ваших изменений и изменений, внесенных другим разработчиком (Merge). Если вам повезло и изменения были сделаны в разных местах файла, то они будут объединены VCS автоматически. Если же нет, вам придется вмешаться. В таком случае VCS при регистрации и извлечении выдает соответствующее оповещение и затем идентифицирует в тексте места, в которых есть несовместимые изменения:

```
<<<<<<< .mine
.wrapchart {
.chartwrap {
.r550
```

В данном случае произошел конфликт между маркерами `<<<<<<<` и `>>>>>>>`. Локальная версия отмечена `.mine` и удалена системой VCS из версии 550 с помощью `=======`. В этом случае оба разработчика изменили имя CSS-класса.

О том, как разрешать конфликты, вы прочтете в следующей главе.

Если существует проблема, вы можете проанализировать временную характеристику изменений и определить, кто ответственен за эти изменения (история). В истории блоками будут перечислены все версии (в соответствии с номером версии) с коммит-датой и коммит-комментариями. В Git это выглядит примерно так:

```
commit da5b01220ce22e5f7e7dd4db58ebdde95158b753 Author: Johannes Jander
johannes@jandermail.de Date: Tue Apr 16 23:55:56 2013 +0200
English localization

commit c6f481a1cdc429875668786a17d2c9b67ff8d7a7 Author: Johannes Jander
johannes@jandermail.de Date: Wed Apr 10 21:53:50 2013 +0200

- flag to mark threads as visited (for the history)
- make images in ThreadView react to clicks
- Adjust target SDK version to 8 again to stop Activities from restarting
when the device orientation changes
```

Прежде всего вы видите уникальный идентификатор (ID) каждого коммита, затем — автора и в следующей строке — дату. За пустой строкой следует соответствующий коммит-комментарий, а за следующей пустой строкой идет блок предыдущего коммита.

Для того чтобы вернуться к определенному состоянию, используется отмена (Revert). Локальное состояние будет отклонено, и из VCS будет извлечена более ранняя версия. Как правило, она новейшая, но возможен вариант и более ранней версии.

Разрешение конфликтов

Когда двое работающих над проектом вносят изменения в одни и те же файлы, а затем пытаются их зафиксировать, в системе контроля версий возникают конфликты. VCS указывает, что изменения были внесены одновременно в один и тот же файл. В лучшем случае изменения встречаются в разных местах текста, тогда система контроля версий может слить изменения. Но в большинстве случаев это изменения в одной и той же строке кода, которые необходимо устранять вручную.

Если изменения встречаются в одной и той же строке, то происходит следующее: тому, кто первым попытался зафиксировать изменения, повезло, и его изменения будут приняты, а второй увидит неприятное извещение об ошибке. В качестве примера приведем файл `ticker.css`, который обслуживается SVN и в котором имя CSS-класса было изменено одновременно двумя разработчиками:

```
>svn ci -m "css classname changed"
Отправляю am/assets/css/ticker.css
Перемещаю данные.svn: E160028: Передача не удалась (Подробности следуют):
svn: E160028: File '/Althorp-Menzies/am/assets/css/ticker.css' is out of
date
```

Да уж, `E160028` не внушает оптимизма — что это может быть? Вот еще одно обновление:

```
>svn up
C am/assets/css/ticker.css
Конфликт в файле "am/assets/css/ticker.css" обнаружен.
Выбор: (p) отложить решение, (df)отобразить изменения,
       (e) редактировать файл, (m) слияние,
       (mc) своя сторона конфликта,
       (tc) чужая сторона конфликта, (s) отобразить все параметры:
```

`C` — это недобрый знак, свидетельствующий о том, что возник конфликт, который должен быть решен при участии разработчика. С помощью команды `p` вы можете в этом месте еще ненадолго отложить решение, но все равно придется реагировать и решать — это касается не только связей, но и работы с системой контроля версий. Если вы уверены в правильности своих изменений, то можете передать их коллегам с помощью команды `mc`. Если же вы предпочитаете принять работу коллег, это можно сделать с помощью команды `tc`.

Как правило, необходимо взглянуть на конфликт, чтобы оценить, что позволит слить изменения. В таком случае вы выбираете команду `m` и получаете представление, в котором противопоставляются свои и чужие изменения:

```
(1) чужая версия (6 строка) |(2) своя версия (6 строка)
----------------------------+-------------------
.chartwrap {                |.wrapchart {
----------------------------+-------------------
Выбор: (1) применяет чужую версию, (2) применяет свою версию,
       (12) применяет сначала чужую версию, затем свою,
       (21) применяет сначала свою версию, затем чужую,
       (e1) редактирует чужую версию и применяет результат,
       (e2) редактирует свою версию и применяет результат,
       (eb) редактирует обе версии и применяет результат,
       (p) откладывает решение конфликта для этого отрезка
           и устанавливает кофликтующие маркеры,
       (a) прерывает слияние файла и возвращается к главному меню:
```

В данном положении, вероятно, будет целесообразно с помощью команды `e1` развернуть чужую версию, ввести собственные изменения вручную и затем сохранить. SVN сразу оживляется:

```
Соединение закончено "am/assets/css/ticker.css".
Выбор: (p) решить позднее, (df) показать изменения,
       (e) редактировать файл, (m) слияние,
       (r) отметить как решенное, (mc) своя сторона конфликта,
       (tc) чужая сторона конфликта, (s) показать все параметры:
```

В нашем примере только один конфликт, поэтому соединение закончено, и вы с помощью команды `r` могли бы уведомить SVN, что выполнили свою работу.

Какую систему контроля версий выбрать?

В настоящее время существует огромное количество различных систем контроля версий. Разброс вариантов велик, от полюбившихся, но устаревающих систем вроде CVN до относительно новой Git. Также существуют платные проприетарные системы, к примеру, ClearCase от компании IBM, Visual SourceSafe, разработанная компанией Microsoft, и системы с открытым исходным кодом.

Если вы не связаны соглашением с фирмой по поводу какой-то определенной системы контроля версий, выбирайте между двумя наиболее популярными сегодня Subversion/SVN (свободная централизованная система контроля версий) и Git, и даже если вы столкнетесь с какими-либо проблемами, в Интернете найдете большое количество полезных ссылок. Если вы работаете индивидуально, платные варианты однозначно проигрывают — даже IT-компании пользуются ими скорее по привычке.

Для начинающих программистов разница между Git и Subversion довольно незначительная. Subversion — система контроля версий, появившаяся в начале двухтысячных годов и впоследствии получившая широкое распространение. Git появилась в 2005 году и благодаря своей службе веб-хостинга github.com очень быстро

стала популярной. К моменту выхода оригинала этой книги Git уже в основном заменила Subversion как в среде профессиональных разработчиков, так и в кругах энтузиастов. На некоторых сайтах вы можете увидеть характерную ленточку с надписью Fork me on github. Это показатель того, что автор управляет содержимым с помощью Git.

Обе системы контроля версий довольно хорошо усовершенствованы, и при выборе одной из них внимание со стороны разработчиков ПО не должно иметь решающего значения — возможно, вам поможет сравнение их преимуществ и недостатков.

Subversion

Существует центральный Subversion-сервер, которым вы можете управлять самостоятельно. В качестве альтернативы можете заказывать готовые хостинг-планы Subversion, которые являются бесплатными для проектов с открытым кодом. Но если не хотите открывать код, придется заплатить.

Каждый коммит загружается напрямую в репозиторий Subversion-сервера и доступен всем участникам проекта.

Каждый проект в Subversion имеет основную линию разработки — так называемый ствол (trunk) и ветви (branches), которые в определенный момент отходят от основной линии разработки. Ветви необходимы для того, чтобы опробовать новые идеи. Если идеи окажутся успешными, они реинтегрируются в ствол, если неудачными — останутся на том же месте, а программист продолжит работу над основной линией разработки.

Git

Существует центральный Git-сервер, управление которым вы можете осуществлять самостоятельно. Как вариант, можете позволить себе управлять проектами с помощью планов, работающих на хостинге, как в случае с Subversion-хостингом, планы для проектов с открытым кодом бесплатные. Особенность Git заключается в том, что, кроме центрального репозитория, на Git-сервере существуют локальные репозитории на компьютере каждого члена команды. И коммиты загружаются не на центральный сервер, а в локальные репозитории. Впоследствии все коммиты с помощью команды Push будут загружены на центральный сервер.

Преимущество такого двухуровневого размещения в том, что некоторые рабочие операции в зависимости от обстоятельств могут быть зафиксированы отдельно. Когда все готово, вы доставляете (Push) на сервер все изменения. Это благоприятствует небольшим коммитам, так как с ними гораздо легче возвращаться назад. В отличие от SVN коммиты загружаются в общий репозиторий не напрямую, что исключает возможность попадания туда случайных ошибок членов вашей команды. С помощью Git вы можете фиксировать даже малейшие изменения и в случае возникновения проблемы вернуться назад, а после успешной отладки предоставить результаты своей работы в распоряжение других разработчиков. Но в то же время, пользуясь Git, вы можете забыть совершить не только коммит, но и Push, тогда

сделанные вами изменения так и останутся на локальном диске, а в худшем случае, если прольете чай на ноутбук, будут утрачены.

По своей организации Git гораздо более гибкая система. В ней нет обязательной основной линии разработки проекта, а многочисленные ветви поддерживаются системой и являются частью Git-культуры. Возможно также одновременное существование равноправных репозиториев. Каждый локальный репозиторий (как правило, у каждого разработчика), в свою очередь, может иметь ветви, в которых разработчик может опробовать свои идеи. Для того чтобы разработчик не тратил огромное количество времени на слияние этих ветвей, Git предлагает очень эффективный инструмент rebase, с помощью которого локальная ветвь может быть синхронизирована с основной ветвью, если разработка в ней уже ушла достаточно далеко вперед. Можно пользоваться Git по тому же принципу, что и Subversion, — наравне с локальными вести центральный репозиторий.

Но такая гибкость имеет свою цену. Git производит впечатление несложной системы, но, когда возникают проблемы конфликта, может оказаться довольно сложно эти проблемы устранить, а затем загрузить в текущий репозиторий.

Полезные идеи при работе с системой контроля версий

Основное правило гласит: за каждым коммитом следует обновление (Update), благодаря этому изменения, зарегистрированные другим разработчиком, являются очевидными. Это тот случай, когда изменения сначала приняты в собственный код и отлажены, а затем зафиксированы, и все работает. Для того чтобы проверить, все ли работает, лучше всего воспользоваться автоматизированным модульным тестированием (Unit-Tests), в главе 16 описано, как это следует делать.

Некоторые системы контроля версий требуют коммит-комментариев, другие позволяют обойтись без них, но тем не менее каждый коммит должен снабжаться таким комментарием, даже если в нем записано: «Добавлен файл personen.html, который мы забыли включить в последний коммит».

Как правило, изменения фиксируются не беспорядочно. Наиболее подходящее время для коммита — то, когда реализована возможность или устранена ошибка, не раньше (изменения будут неполными), но и не позже (возможны два изменения в одном коммите). Так как в Git и других системах контроля версий есть локальный репозиторий, многие авторы советуют чаще выполнять коммиты, так как новые VCS работают довольно быстро и один коммит практически не прерывает поток мыслей разработчика. А если к каждому коммиту будет написан комментарий — что мы настоятельно рекомендуем делать, — то не имеет смысла проводить фиксацию слишком часто, чтобы не отвлекаться на составление внятного комментария, когда вы размышляете над кодом.

Перед коммитом кода просмотрите еще раз все закомментированные блоки кода и удалите их. После этого проверьте, являются ли комментарии к коду до сих пор актуальными. В случае сомнения поможет утилита diff, которая еще раз обобщит все сделанные вами изменения — наглядные (когда возможно прочитать

патч) и неочевидные (когда невозможно прочитать патч). В главе 20 описаны diff-подобные инструменты с графическим пользовательским интерфейсом, компенсирующие большинство неудобств, связанных с diff. Если вы не боитесь связываться с командной строкой, то можете позволить себе отображать вывод diff прямо через нее.

Зарегистрируйте все данные, которые принадлежат проекту. Если спустя некоторое время решите продолжить работу над проектом, кроме исходного кода, вам понадобятся и другие файлы. Что делать, если имеющиеся у вас данные испытаний (результаты отладки) настолько большие, что превышают 1 Гбайт? Зарегистрируйте их вместе с кодом! Вы пользуетесь базой данных? Экспортируйте схему базы данных и ее содержимое как язык описания данных (DDL) или Dropskript и зарегистрируйте ее. Тогда в случае, если сервер базы данных будет изменен или база данных выйдет из строя, с помощью скрипта вы за пару минут сможете восстановить схему.

DROPSKRIPT

Dropskript — это текстовый файл с SQL-операторами, с помощью которых схема базы данных может быть восстановлена со всеми таблицами и при необходимости с содержанием.

Неудачные идеи при работе с системой контроля версий

Наиболее распространенная ошибка при работе с VCS — это редкие обновления рабочей версии. Опасность заключается в том, что в течение дня человек не интересуется, были ли внесены изменения другими разработчиками, но тем не менее фиксирует собственные результаты. Если в итоге в файле возникает конфликт, то VCS откажется от такого коммита. В таком случае, прежде чем снова совершить коммит, необходимо локально устранить конфликт.

Часто при совершении коммита разработчик слишком долго колеблется. Если со времени последнего коммита прошло три дня, а вместе с вами над кодом работает кто-то еще, то вероятность конфликта довольно высока. Проблема похожа на описанную ранее, но в данном случае разработчик получает предупреждение о конфликте, когда забирает всю ветвь. Также в этом случае у вас есть тот, кто слишком долго ждал и несет за это ответственность: ему придется локально устранить конфликт, прежде чем вы сможете продолжить работу.

Еще один известный рецепт неприятностей — коммиты «в последнюю минуту», когда до наступления крайнего срока хочется успеть сохранить еще одну возможность. И если в этот код закралась ошибка, раздражение вполне справедливо, потому что она попадает ко всем коллегам, а возможно, и в готовый продукт.

Существуют данные, которые не обслуживаются системой контроля версий. Файлы настроек, в особенности те, которые содержат путь, имя пользователя и пароль, должны оставаться на локальном компьютере, особенно если вы все программируете самостоятельно. Если же система контроля версий доступна

для других, вы должны забыть про пароль. Проблема в том, что однажды зарегистрированный пароль сохраняется в истории системы контроля версий.

Система контроля версий — элемент ПО

Система контроля версий может использоваться не только как средство управления исходным кодом, но и в качестве базы данных, что позволяет управлять программными данными.

При внесении клиентов в командную строку VCS позволяет создать сценарий. Практически на любом языке программирования можно вызвать внешнюю программу. Для этого существует команда, которая выглядит примерно как `system()`, она описана в главе 22.

Если вы создали новый файл `201 20903.txt`, с помощью команды `system('svn add 201 20903.txt')`, за которой следует другая команда `system('svn commit -m ""')`, этот файл может быть загружен в Subversion-репозиторий. Данная операция может быть выполнена как с локального компьютера, так и с сервера. Если, например, вы сейчас пишете программу для обработки данных, то с помощью двух этих строк можете переместить свои результаты на удаленный сервер, откуда посредством `system('svn up')` они будут регулярно извлекаться и вноситься на веб-сайт.

Это кажется довольно просто, но без системы контроля версий или базы данных придется самостоятельно программировать примерно следующее.

1. ПО должно в течение нескольких минут в режиме онлайн предоставить доступ к результатам обработки на веб-сервере beispiel.test.
2. Передача результатов на веб-сервер beispiel.test должна быть зашифрована.
3. Все выведенные результаты должны быть заархивированы как на локальном компьютере, так и на сервере.
4. Обработка должна проходить автоматически через регулярные промежутки времени. Передача данных должна происходить даже без вмешательства разработчика.

Для того чтобы с самого начала самому все это реализовать, вам понадобится немало времени. Subversion (либо любая другая SVN) обеспечивает зашифрованную передачу данных по ssh, удобна для работы со сценариями, а также это идеальный инструмент, который можно относительно легко встроить в собственную программу.

Как сервер для хранения данных система контроля версий, в сравнении с базой данных, имеет ряд преимуществ.

- Автоматическое архивирование, так как VCS, как правило, создает версии вводимых данных. А обновление в базе данных ведет лишь к переписыванию данных.
- Если имеется информация, которую можно представить и как документы, и как отдельные множества данных, человек не может воспользоваться преимуществами реляционной базы данных, так как она применяется лишь как место

хранения данных. В такой ситуации для облегчения жизни можно с равным успехом использовать систему файлов или VCS.

Но в то же время система контроля версий как сервер хранения данных в сравнении с базой данных имеет свои недостатки.

- В базе данных поиск записи проходит довольно быстро и гибко, VCS — это хранение решения.
- Если необходимо сохранить данные в иерархическом порядке, база данных благодаря связям и таблицам предлагает гораздо больше возможностей.

В большинстве случаев системы контроля версий не могут заменить базы данных, но для элементарного блога или системы редактирования, которые фиксируются только как отдельные страницы с датами внесения изменений и последним редактором, VCS могут быть вполне эффективным машинным интерфейсом.

22 Command and Conquer: из жизни командной строки

> Если вам требуется сделать больше пяти кликов, научитесь пользоваться командной строкой.
>
> *Шон Андерсон, blog.AdminArsenal.com*

Командная строка — это не вымерший динозавр времен начала обработки данных, а продуманный интерфейс для чрезвычайно ленивых людей. Неопытному программисту с первого взгляда трудно с этим согласиться. Но если внимательнее присмотреться, то в командной строке могут быть скрыты простые решения проблем, которые часто встают перед нерегулярно работающими программистами.

Командная строка и ее средства берут свое начало с тех времен, когда не приходилось даже думать о графических пользовательских интерфейсах, а пользователи должны были быть экспертами во всем и потоки работ были еще сравнительно простыми. Из-за того что компьютеры работали медленно и объем памяти был небольшим, основные задачи электронной обработки данных в то время сводились к повторяющимся (циклическим) отчетам, расчетам заработной платы, обработке лог-файлов. Все это можно было отлично решить с помощью небольших программ, которые зачастую были связаны другими способами через сценарии — этот аспект остается весьма значимым и до сегодняшнего дня.

После недолгого периода ознакомления вы сможете ежедневно перемещать файлы и копировать каталоги в оболочку гораздо быстрее, чем в графический пользовательский интерфейс оперативной системы. Кроме того, все действия регистрируются, являются воспроизводимыми (повторяемыми) и поэтому легко автоматизируются.

Работа с классической командной строкой Unix не только повышает эффективность в общем, но и способствует улучшению дизайна ПО: многие программы являются хорошим примером модульного исполнения, они оптимизированы для совместной работы с другими программами. Вместо того чтобы пытаться кое-как решить большое количество проблем, эти программы делают акцент на оптималь-

ном решении небольшой проблемы, и больше ничего. Все остальное передается другим программам. Это в какой-то степени соответствует разделению труда между различными классами в области объектно-ориентированного дизайна ПО. К тому же большинство этих средств основываются на библиотеках, которые могут использоваться этими же программами. Если вы знакомы с этими программами, вам будет гораздо сложнее вновь ненароком изобрести велосипед и вы получите начальное представление о работоспособности библиотеки. Например, тот, кто разобрался со средством openssl и догадался, как можно внедрить криптографию в собственную программу, использовал для этого именно библиотеку, на которой основывается данное средство.

Повышение эффективности с помощью автоматизации

Введение в эксплуатацию конвейера в автомобильной промышленности привело к тому, что на замену довольно разносторонней, интересной, позволяющей получить удовлетворение автомобилестроительной деятельности пришло большое количество рабочих мест, на которых каждый работник постоянно изготавливал лишь одну деталь. Это первый этап автоматизации — разделение на атомы, мелкие постоянные наборы идентичных действий, выполняемых в строгой последовательности, который обусловил второй этап — введение в эксплуатацию программируемых роботов/автооператоров, выполняющих без потери рассудка скучные задания гораздо надежнее, чем любой человек. На сегодняшний день в автомобилестроении, грубо говоря, дизайнеры и программисты заменены роботами. Две задачи (профессии), которые, как правило, являются довольно разносторонними, интересными и позволяющими получить удовлетворение.

Конвейер в автомобилестроении можно сравнить с командной строкой в разработке ПО: она не делает жизнь красивее и ярче, но позволяет сделать следующий шаг в эволюции в направлении эффективности через автоматизацию. Командная строка переносит поле деятельности разработчиков ПО с конкретного на абстрактное, то есть на то, что они делают охотнее всего, — на программирование!

Два программиста, два мнения

Когда я думаю о том, что сделало меня неплохим программистом, то прежде всего мне приходит на ум переход от графического пользовательского интерфейса к средствам командной строки как к основному используемому мной в программировании интерфейсу «человек — машина».

При осуществлении этого шага я довольно долго мешкал. Несмотря на то что я уже в 80-е годы в дочернем отделении компании Commodore Amiga напрямую столкнулся Unix-подобной оболочкой, переход на систему DOS, а в конечном счете на Windows надолго отбил охоту переходить на использование командной строки: непостоянство и яркий ошибочный дизайн привели к тому, что использование «командной строки» в Windows до сих пор является разочаровывающим опытом.

Прошло больше 20 лет, прежде чем я, выполняя новое задание, заметил, как мои коллеги по команде оперативно создавали в своих оболочках аппаратные среды, последовательно автоматизируя то, что было ранее сделано эффективными разработчиками. И я поспешил наверстать упущенное. К моему счастью, в этой компании практиковалось парное программирование, когда у вас постоянно есть напарник, который каждый день меняется. Мои напарники проявили снисхождение к новичку и научили меня большому количеству приемов, которые позволили быстрее достигать цели и проводить день за более приятными занятиями, чем навигация по проводнику Windows. Мои коллеги, естественно, работали параллельно на всех трех платформах (Linux, Windows и Mac OS X) и чувствовали себя за любым компьютером как дома, потому что у всех была одна командная оболочка bash-Shell: для систем OS X и Linux — домашняя, а для нового компьютера с системой Windows самым обычным действием была установка cygwin. После дооборудования Unix-подобной среды такие устройства стали годны к употреблению.

Ян Бёльше

У меня командная строка нередко вызывает плохое настроение. Что меня сделало неплохим программистом, так это IDE-интерфейс, такой как Intellij IDEA, который понимает код и говорит: «Эта ветвь кода не выполнена, просто уйди».

С моей точки зрения, IDE-интерфейс с хорошим сочетанием клавиш (клавишами быстрого вызова) функционирует значительно лучше, чем shell-сценарий, но когда, не меняя контекст, необходимо из одной программы сделать другую, от культурного шока графический пользовательский интерфейс молча переходит к командной строке.

Тем не менее я достаточно реально смотрю на вещи, чтобы не видеть преимуществ командной строки: я администрирую свой небольшой арендуемый Linux-сервер через ssh-протокол, и с моей системой контроля версий я работаю над проектом иногда через IDE-интерфейс, иногда через командную строку. Мой мозг гораздо лучше воспринимает понятия через детали, поэтому в школе у меня были плохие отметки по латинскому языку и хорошие — по математике. Сейчас, к сожалению, это приводит к тому, что мне приходится постоянно уточнять параметры многих shell-команд.

Ганс

Когда вы можете предусмотреть наличие определенных средств, например, когда пишете cgi-сценарий (общий межсетевой интерфейс) для сайта и сами администрируете сервер, тогда знание этих средств так же важно, как знание стандартной библиотеки того языка, на котором вы программируете. Краткая командная строка, которую вы вызываете для своей программы, может помочь сэкономить сотни строк кода и выполнить задание более эффективно и понятно для других, чем это сделает ваш собственный код.

В крайнем случае эти знания означают, что вы не впервые приступаете к этому проекту, так как комбинация из двух или трех средств уже дала конкретный результат, который вы хотели получить. Ведь самым лучшим и гарантированно отлаженным считается код, который написан не впервые. Поэтому лучше и профессиональнее тот программист, который быстрее понимает, что программиро-

вать уже нечего. А выгоду можно получить, хорошо разобравшись с командной строкой[1].

Наши длиннобородые предки

Интерпретатор командной строки — это программа, которая принимает и выполняет команду в форме текста, и, как результат, в дальнейшем команда представлена в виде текста. Это особенность интерактивного режима интерпретаторов языков программирования, таких как Python, Ruby или TCL. Консоль Java-Script в Firefox-плагине Firebug функционирует по такому же принципу, который более подробно описан в разделе «REPL» главы 20.

Поскольку изобретатели Unix были людьми ленивыми на рекомендации, называют этот интерпретатор sh (сокращение от shell), а программы, которые он выполняет, соответственно shell scripts (shell-сценарии). Программировать на этом языке совершенно непривычно. Мы можем отговорить от этого, потому что синтаксис, например, таких управляющих структур, как `if` и `switch`, или контур/цикл программы абсолютно другие и не являются хорошими и интуитивными. Если речь идет о коротких задачах, чаще пользуются такими языками, как Perl, Python или Ruby. Интерактивный режим оболочки, напротив, очень полезен.

Существует целый ряд различных оболочек Unix, например C-Shell (называются также csh или tcsh), Korn Shell (ksh) или Z Shell (zsh). Хотя они и не считаются новейшими, на сегодняшний день Bourne Shell является наиболее часто используемой[2]. Эта оболочка была названа так в честь своего разработчика Стивена Борна, который в 1970-е годы был сотрудником компании AT&T.

То, что люди иногда используют название «оболочка», иногда «терминал», а иногда «консоль», имеет главным образом историческую причину, в настоящее время эти термины по сути означают одно и то же. Shell (оболочка) — это настоящая программа, название которой происходит от того, что первоначально она действовала как оболочка операционной системы и была сверху, а сегодня работает на уровне, расположенном глубоко внутри. Терминал и консоль получили такие названия из-за того, что раньше они были самостоятельными устройствами — экраном и клавиатурой, которые

---

[1] Как правило, человек избегает таких неприятных ситуаций, в которой оказалась программистка Хизер Артур в начале 2013 года. Она создала для личного пользования средство командной строки, которое назвала replace и разместила как открытый исходный код на GitHub. Вызвал насмешки в «Твиттере» тот факт, что свойства уже существующих средств командной строки sed и find она заменила чем-то более сложным. Подробное обсуждение проблемы и возможные альтернативы вы можете найти по адресу news.ycombinator.com/item?id=5107491.

[2] Если быть точными, то чаще всего сегодня используется доработанная копия оболочки, придуманной Борном, название сохранено из-за необходимости соблюдать авторское право. Эта копия является частью GNU-проекта, который по бесплатной лицензии предоставляет в пользование все наиболее важные Unix-средства, называется bash, что расшифровывается как Bourne-again Shell и является хорошим примером любви разработчиков Unix к игре слов.

позволяли осуществлять интерактивную коммуникацию с компьютером (процессором) без всяких перфокарт! Сегодня эти устройства уже стали программным обеспечением, а понятие «терминал» для Mac или, например, Ubuntu-Linux существует только в окне графического пользовательского интерфейса, для того чтобы дать представление о нижележащих уровнях операционной системы.

Windows

Среди операционных систем Windows занимает особое положение. Если Mac OS и Linux основаны на Unix, то Windows, для того чтобы применить все средства, упомянутые в этой главе, придется либо пройти свой собственный путь, либо установить Unix-подобную среду cygwin.

Если вы разрабатываете ПО и хотите, чтобы оно работало в системе Windows, в вашем распоряжении три интерфейса командной строки.

- DOS-среда, которую ласково называют «приглашение к вводу». Немногие работают с этой средой, потому что ее возможности довольно ограниченны. Правда, можно написать .bat-сценарий, для того чтобы кое-что автоматизировать, но все же среде DOS недостает многих практических средств.
- Windows PowerShell, которая совершенно отличается по своей идее от среды командных строк и сценариев в оболочках Unix тем, что является объектно-ориентированной. Ее средства, называемые cmdlets, могут быть настолько связанными, что вывод данных одного является вводом для другого. Это похоже на философию командных строк в Unix, но cmdlets не передает никакого текста, только .NET-объект. Сценарии PowerShell могут считывать также стандартные Windows-программы, для этого привлекается традиционный язык программирования, такой как Ruby или Perl. PowerShell — довольно интересная концепция, но она выходит за рамки общепринятого использования командной строки, которое мы можем рассмотреть в этой краткой главе. Если вы работаете исключительно с Windows, вам необходимо более подробно рассмотреть PowerShell.
- cygwin. cygwin открывает пользователям Windows целый мир средств командной строки с открытым кодом. После установки стандартного пакета вы можете без всяких вспомогательных инструментов переустановить содержащиеся в управляющем пакете базы данных, веб-серверы, языки программирования и в результате получите возможность работать в Windows так же, как в Unix. Недостатком является то, что вы не сможете использовать специальные операции Windows, как в случае с PowerShell. А преимущество в том, что вам не понадобятся специальные знания и вы сможете одинаково успешно работать как на компьютере с Windows, так и на Linux-сервере.

Что следует знать каждому программисту

Пользователи, нерегулярно работающие с командной строкой, могут оказаться в замешательстве из-за того, что некоторые команды, например, `cd`, встроены в оболочку и могут быть запущены из любого места, а другие, так как являются небольшими

автономными программами, имеют собственное местонахождение. Обычные места для размещения таких частей оболочки — `/bin`, `/usr/bin`, `/usr/local/bin`. В большинстве случаев в повседневной жизни несущественно, встроена команда в оболочку или нет, но не удивляйтесь, если появляются команды, которых нет в указанных каталогах и которые не могут быть найдены с помощью команд `find` или `locate`.

Если был запущен процесс с большим объемом текстовых данных или данный процесс не отвечает, его можно прекратить с помощью Ctrl+C. Но этот прием срабатывает не для всех команд: если после запуска необходим ввод каких-либо данных, прекратить процесс поможет Ctrl+D. Если ничего не помогает, можно закрыть текущее окно и открыть новое.

Параметры

В графическом пользовательском интерфейсе для похожих версий одних и тех же команд либо существуют разные кнопки, либо необходимо установить различные флажки. В Shell эта же задача решается с помощью следующих команд и параметров:

- `ls`. Показывает файлы, находящиеся в `working directory` (рабочем каталоге), но исключает скрытые файлы. (В системе Unix файлы считаются скрытыми, если их имя начинается с `.`.) В системе Windows такая команда известна как `dir`;
- `ls -a`. Отображает все файлы, находящиеся в `working directory`. Параметр `a` означает, что следует показывать все файлы, даже скрытые;
- `ls -la`. Показывает все файлы и форматирует результат в виде списка (параметр `l`).

Последовательность параметров ни на что не влияет, `-la` функционирует так же хорошо, как `-al`. Оба параметра, по сути, являются лишь краткой формой от `ls -l -a`. Зачастую, когда количество букв ограничено, параметры с различными командами имеют различное значение, например, `-i` может означать «интерактивный» либо «нечувствительный к регистру символов».

К сожалению, при работе с параметрами существуют различные условности. Одной из наиболее важных представляется так называемая Longopts (которая, к счастью, в большинстве случаев используется только GNU-средствами в системе Linux), в которой параметры — это не отдельные буквы, а целые слова. Некоторые Longopts должны быть записаны только раздельно и не могут быть записаны слитно. Longopts обозначаются `--`:

```
rsync -v является тем же, что и rsync --verbose
```

В графическом пользовательском интерфейсе множество различных кнопок, ссылок, параметров, которые занимают много места на экране и которые не всегда легко отыскать в меню и подменю. Но бывают случаи, когда человек не видит то, что ему нужно. Командная строка скрывает наличие большого количества возможностей. С помощью `-h`, `-help` или `--help` открываются вспомогательные страницы с основными параметрами для данной команды.

С помощью параметра `man` (manual) вы можете вывести детали использования каждой команды напрямую из командной строки. Она применяется в том случае,

если вы знаете имя команды, но не знаете точно, что под ним подразумевается: например, `man grep`. Кроме того, менее ловким пользователям командной строки придется привыкнуть к навигации в этом тексте. Вы можете с тем же успехом использовать для поиска manpage grep ваш поисковик, но результат будет таким же. К сожалению manpage не содержит достаточного количества примеров и пригодна лишь для настоящих профессионалов, которые, в принципе, все знают и хотят лишь слегка освежить знания о синтаксисе команд. О каждом средстве командной строки, естественно, имеется статья в «Википедии», в которой все доступно изложено и приведены примеры их использования. Кроме того, в конце данной главы вы найдете ссылку на источник, где все подробно объясняется и есть много примеров использования средств командной строки.

Аргументы

Наряду с параметрами многие программы командной строки учитывают и аргументы. Параметры влияют на то, как непосредственно работает команда, а аргументы указывают на источник и/или цель/объект, например файл. Команда `cp` копирует существующий файл и дает копии следующее имя:

```
cp источник.txt объект.txt
```

Многие программы также принимают некоторые аргументы для того, чтобы одинаково реагировать на все, например, `rm` удаляет файлы:

```
rm файл1.txt файл2.txt
```

Здесь вас поджидает много неприятностей, если вы используете имя файла с пробелом:

```
rm имя файла пробел.txt
```

Обычно `rm` выдает ошибку, если не находит файлы `имяфайла` и `с пробелом.txt`. Но вы можете предотвратить потерю данных, если запишете вместо этого:

```
rm "имя файла с пробелом.txt"
```

Как вызывать команды командной строки на разных языках программирования

В дальнейшем в данной главе мы рассмотрим Shell-команды только с той точки зрения, что сопряжение таких команд позволит упростить выполнение часто повторяющихся задач, так как они являются наиболее распространенными приложениями при разработке ПО. Для того чтобы сократить свою работу, вы можете воспользоваться языками программирования Shell-команд.

В качестве примера приводится команда, которая звучит как system():

- Ruby — system 'ls', '-l', '/usr/bin'
- Python — cmd =‚ls -l /usr/bin‘ os.system(cmd)

- Perl — my $cmd = "ls -l /usr/bin"; my $status = system($cmd);
- PHP — $cmd = 'ls -l /usr/bin'; $status = system($cmd, $retval);

Вместо ls вы можете, конечно, вызвать любую другую программу командной строки, например cvn (Subversion-Client) или scp (копирует файлы с одного сервера на другой).

Что делать с результатами?

Если человек больше ничего не дописывает, результат команды обычно появляется в окне терминала. Если вместо этого он должен выгружаться в какой-нибудь файл, можно изменить путь с помощью >:

```
ls >all_my_files.txt
```

Это самая простая из прочих возможность писать лог-файлы. С каждым вызовом этой команды `all_my_files.txt` будут перезаписаны. >> добавляет результаты в конец имеющегося файла (и создает этот файл, если его не существует).

< берет содержимое файла и делает его данными ввода программы. С помощью < (читает из файла) и > (пишет в файл) можно создать отличные программы фильтрации, чтобы фильтровать и сортировать труднообозреваемые объемы текста.

Традиционно существуют два потока результатов: `STDOUT` (standard output) — это место, куда должны выгружаться стандартные результаты команды, `STDERR` (standard error) выдает случайно выгруженные ошибки. Оба потока для простоты пронумерованы: `STDOUT` — `1` и `STDERR` — `2`. Если вывод данных осуществлен в файл без сообщения об ошибке, тогда это выглядит так:

```
echo test 1>testfile.txt
```

Чтобы изменить путь `STDERR`, необходимо заменить `1` на `2`. `2>&1`, что вы, вероятно, довольно часто видели, означает: `STDERR` должен быть перенаправлен на `STDOUT` (`&` сообщает оболочке о том, что ничего, кроме файла с неблагоприятным именем `1`, не подразумевает).

Вместе с тем существует еще и `STDIN` под номером `0`, это стандартные данные ввода, например, через клавиатуру. Но когда файл с помощью < перенаправляется на `STDIN`, перенаправляется только его содержимое.

Сопряжение результатов

Основная причина того, что командная строка так любима своими почитателями, состоит в том, что все команды, как детали конструктора «Лего», могут присоединяться друг к другу. Каждое средство выполняет одну и только одну задачу, но настолько хорошо и универсально, как только возможно. Сложные задачи приходится решать в том случае, когда результат одной команды является данными ввода для другой команды. Это происходит через Pipe-оператор |:

```
ls -l | less
```

Если полученный список файлов оказался длинным и неудобным для использования, воспользуйтесь `less` — командой, которая обычно только заполняет экранную страницу (см. далее).

Типичными деталями «Лего» командной строки, которым будут переданы результаты для обработки, являются `sort` (сортирует результаты, например, по алфавиту), `grep` (возвращает только к тем строкам, которые содержат определенную последовательность символов), `uniq` (отбрасывает все строки, которые встречаются дважды), `head` и `tail` (указывают на начало и конец файла соответственно). Примеры их использования приведены далее. Конечно, многие из этих так называемых деталей зависят друг от друга:

```
sudo tail -n 100 /var/log/apache2/access.log | grep ' 404 ' | less
```

Эта команда просмотрела последние 100 строк журнала веб-сервера на предмет доступа к несуществующим страницам и выдала постраничный результат. При больших лог-файлах процесс может проходить быстрее, так как `grep` разблокирует все файлы.

Символы-джокеры

Многое из того, что при других обстоятельствах можно выполнить в графическом файловом менеджере или SFTP-клиенте, получается лучше с помощью командной строки, так как можно более точно указать, к каким файлам можно обращаться. В этом помогут так называемые символы-джокеры. В табл. 22.1 приведены наиболее распространенные символы-джокеры.

Таблица 22.1. Наиболее распространенные символы-джокеры

Символ-джокер	Значение
*	Соответствует любым символам
f*	Обнаруживает все файлы, имена которых начинаются с «f»
f*.txt	Обнаруживает все текстовые файлы, имена которых начинаются с «f»
?	Соответствует только одному символу
f??sch.png	Обнаруживает frosch.png, но не fisch.png
[abc] oder [A-Z]	Соответствует группе символов
[abc]*	Обнаруживает все файлы, имена которых начинаются с «a», «b» или «c»
[A-Z]*	Охватывает все файлы, имена которых начинаются с прописной буквы «A» и заканчиваются прописной буквой «Z» (будьте внимательны, сюда не включены умляуты, для того чтобы включить их и корректно использовать различные письменные системы (шрифты), вам следует разобраться с Character Classes [[:upper:]])
[!abc] или [!A-Z]	Вызывает противоположность и исключает все символы этого выбора
asdf[!0-9]	Охватывает asdfg, а не asdf3

В различных оболочках символы-джокеры обладают различными возможностями, которые выходят за пределы перечисленных здесь, но те, что приведены в таблице, функционируют повсеместно.

Навигация

Рабочий каталог — это то место, где мы обычно находимся. Например, мы хотим с помощью `ls` направить все файлы в одну папку, тогда команда ссылается — если больше ничего не указано — на местонахождение `.home` — каталога, в котором вы изначально находитесь. Обычно он называется `/home/bernd` или как-то похоже.

Существуют следующие команды для навигации по папкам:

- `pwd` — сокращение от Print Working Directory («вывести рабочий каталог»). Указывает на то, в каком каталоге вы сейчас находитесь. С помощью команды `pwd` профессионалы проверяют, чтобы каждая команда, с помощью которой что-нибудь может быть безвозвратно удалено, находилась именно в том месте, где вы хотите что-то удалить;
- `cd` — сокращение от Change Directory («смена каталога»). С ее помощью вы можете перейти в любую другую папку. Использование `cd` позволит вам (без указания пути) вернуться в свой домашний каталог `/home/bernd/` и каталог на уровень выше, например из `/home/bernd/` в `/home/`. Команда `cd` необходима также для того, чтобы вернуться в каталог, в котором вы были в последний раз. Так можно быстро перейти из одного каталога в другой.

Файлы

Dotfiles (точка-файлы) — это все файлы, имена которых начинаются с точки «`.`», которые не выводятся на экран с помощью стандартной команды `ls` и в большинстве случаев необходимы для того, чтобы сохранить настройки программ командной строки. (С помощью `ls -a` можно увидеть все файлы, даже точка-файлы). `./` и `../` занимают особое положение. `./` — обычно это каталог, в котором вы находитесь, `../` — это каталог на уровень выше.

Рассмотрим команды для работы с файлами.

- `mv`. С помощью команды `mv` (move) можно не только переименовывать файлы, но и перемещать их. В отличие от графического файлового менеджера, в командной строке имеется гораздо больше возможностей для того, чтобы, например, большое количество файлов по определенному критерию найти и переименовать. И когда человек дистанционно получает право доступа к серверу, `mv` справляется с довольно большим количеством файлов гораздо быстрее, чем SFTP-Client, хотя сервер и клиент не должны уведомлять о каждом отдельном файле.
  - `mv file1 file2` переименовывает `file1` в `file2`. Если `file2` уже существует, он будет перезаписан. Неопытным советуем использовать параметр `-i` (interactive), тогда `mv`, прежде чем перезаписать существующий файл, будет запрашивать подтверждение. Параметр `-i` располагается между `mv` и первым именем файла.

- `mv file1 file2 file3 some_dir` переместит три файла в каталог `some_dir`. В этом случае вы также можете использовать параметр `-i`.

- `cp` — сокращение от copy («копировать»). Преимущества в сравнении с файловым менеджером и SFTP-Client такие же, как и у команды `mv`.
  - `cp file1 file2` копирует содержание `file1` в `file2`. Если `file2` уже существует, он будет перезаписан, в этом поможет параметр `-i`.
  - `cp file1 some_dir` поместит копию `file1` в каталог `some_dir`.
  - `cp -r my_important_data_dir some_backup_dir` поместит файлы в новый каталог `some_backup_dir` и скопирует в него `my_important_data_dir` со всеми вложенными подкаталогами. Если каталог `some_backup_dir` уже существует, все скопированные данные под своими первоначальными именами будут помещены в подкаталог или в `some_backup_dir/my_important_data_dir`.

- `rm`. С помощью команды `rm` (remove — «удалить») удаляются файлы и каталоги: `rm file1`.

  Для того чтобы удалить каталог, необходимо воспользоваться командой `rm` с параметром `-r` (recursive). Сначала следует удалить из каталога все файлы, а затем и сам каталог.

  При использовании команды `rm` может случиться непоправимое, поэтому мы советуем, в особенности тем, кто работает с символами-джокерами, соблюдать меры предосторожности от случайного удаления с жесткого диска. Здесь также может помочь интерактивный параметр `-i`, который перед каждым удалением потребует подтверждения:

  ```
  rm -i f*
  rm: remove fisch.png (y/n)? y rm: remove frosch.png (y/n)? n Чтобы не подтверждать отдельно удаление каждого из 100 файлов, можно вписать команду ls, в нашем примере ls f*.
  ```

  Для того чтобы проверить, все ли файлы были удалены, клавишей со стрелкой вверх вызывают команду и `ls` заменяют на `rm`.

- `diff`. Эта команда сравнивает между собой два файла: `diff test1 test2`. Более подробно команда `diff` рассмотрена в главе 20.

Права доступа. Каждый файл снабжен информацией о том, кто должен его прочитать, дописать или выполнить. Это право может быть изменено с помощью менеджера файлов и SFTP-Client, но если необходимо работать одновременно с несколькими файлами, то гораздо быстрее сделать это с помощью командной строки. С помощью `ls -l` можно увидеть, какие права установлены, примерно так:

```
$ ls -l drwxr-xr-x 3 tomcat7 users 3350528 Jul 1021:34 logs
-rw-r--r-- 1 snow snow 470 Dec 182012 post.html
-rw-r--r-- 1 snow snow 1955 Dec 182012 readme.txt
-rwxr-xr-x 1 snow snow 126 Dec 182012 render.sh
-rw-r--r-- 1 snow snow 494 Dec 182012 rendertemplate.htm
```

То, что начинается с `d`, — это каталог; `r` означает read, `w` — write, а `x` — execute. Первое сочетание `rwx` показывает, какими правами обладает пользователь этого файла, второе — права группы пользователей, третье `rwx` — права остальных пользователей по всему миру. Символ `-` указывает, кому доступ запрещен. Иногда на это право указывают с помощью новых букв в восьмеричной системе счисления, что гораздо быстрее и убедительнее. Этой формой человек добивается того, что буквы `rwx` на цифры `421` заменяются и добавляются. `r--` соответствует `4`, `rw-` соответствует `6`, `r-x` — `5`. `777` обозначает: всем можно все.

Пользователем нового файла зачастую является тот, кто его создал. Только этот пользователь (равно как и робот) имеет право изменять данный файл. Это происходит с помощью команды `chmod`:

```
chmod 644 test.txt
```

Это означает, что пользователь `test.txt` имеет право читать и дописывать данный файл, все остальные могут его только читать.

Если вы загружаете сценарии из системы Unix или выгружаете их, то они являются для операционной системы просто текстовыми файлами, которые вы не можете запустить как программу. Для того чтобы сделать их исполняемыми, необходимо добавить:

```
chmod +x meinskript.sh
```

Все PHP-файлы в каталоге можно сделать исполняемыми с помощью команды:

```
chmod +x verzeichnis/*.php
```

`sudo`. Как правило, вы можете изменять только те файлы, которые создали сами, и читать только те, к которым имеете право доступа. Многие системные журналы могут быть прочитаны только администратором (с корнем `User`), но вас, как разработчиков, они интересуют прежде всего тем, что содержат детали часто встречающихся ошибок.

Если вы хотите прочитать эти файлы, например, с помощью `tail`, тогда необходимо перед этой командой написать `sudo`: `sudo tail /var/log/syslog`. Она следит за тем, чтобы команда `tail` выполнялась не напрямую, для начала вам придется ввести собственный пароль. `sudo` универсальна: неважно, перед какой командой вы ее запишете, она будет выполнена на уровне доступа администратора.

Во время веб-разработки, когда многие одновременно могут пользоваться одним компьютером, чтобы избежать неприятностей, следите за тем, чтобы не изменять свой уровень доступа с помощью `sudo`.

Просмотр

Рассмотрим команды для просмотра файлов.

- `less`. Команда `less` относится к тому времени, когда скроллинг еще не был изобретен, но и сегодня может оказаться полезной. Она отражает содержание файла,

или, точнее, данные других команд, всегда только на одной экранной странице, например, так:

```
less logfile.txt
```

В конце страницы стоит двоеточие, которое обозначает ожидание ввода данных. Чаще всего это пустая строка, с помощью которой можно перейти на новую экранную страницу. С помощью `g` осуществляется переход в начало, а с помощью `G` — в конец текста.

- `head`. Показывает только начало текстовых данных вывода, то есть первые несколько *n* строк. Она может использоваться как для файлов, так и для данных вывода других программ командной строки:

  ```
  head -n 20 logfile.txt
  ```

  В данном случае команда выдает первые 20 строк файла `logfile.txt`. Если перед данными вывода использовать `ls`, это будет выглядеть следующим образом:

  ```
  ls -lt | head -n 20
  ```

  Команда показывает список новейших 20 файлов текущего каталога, сортирует их по дате изменения. Параметр `-l` означает «форматировать в виде списка», `-t` — «сортировать по времени», `-n` указывает на количество файлов, которые необходимо показать.

- `tail`. Эта команда является обратной по действию команде `head`, с ее помощью вы можете увидеть конец файла. Она пригодится, если вам в конце большого лог-файла необходимо найти определенную учетную запись.

  Например, для того, чтобы попасть в конец большого Apache-httpd-лог-файла с помощью `less`, вы можете положить пресс-папье на клавишу пробела (чтобы прокрутить файл) и пойти попить кофе. Благодаря команде `tail` вы можете очень быстро увидеть нужные последние строки. В стандартном случае показываются десять строк, но если к команде `tail` добавить параметр `-n 100`, то в результате вы увидите последние 100 строк.

  Если ваше программное обеспечение написано в лог-файле, пригодится также параметр `-f` (follow): `tail -f` покажет вам последние строки файла в динамике. Программа зависит от новых строк файла, которые обновляются на экране с помощью `tail`. Например, `-f access.log` показывает в реальном времени, что происходит на вашем веб-сервере.

- `tee`. Создает T-образный тройник для команды с той целью, чтобы данные выхода можно было одновременно записать в файл и читать, например, так:

  ```
  curl http://example.com/test.html | tee test.html | grep anfang
  ```

  HTML веб-сайта будет записан в файл `test.html`. Одновременно на терминале будут показаны все строки, содержащие слово `anfang`. Команда `tee` может быть полезна, когда запущена долгоиграющая программа, которая выполняет отладку данных вывода. Та же отладка данных вывода пишется в файл, чтобы позднее провести детальный анализ, например, отфильтровать данные вывода по строкам, содержащим ошибку, для того чтобы просмотреть только самые серьезные ошибки.

- `wc`. Команда `wc` (word count) показывает, сколько строк, слов и байтов содержит файл:

  ```
  wc diplomarbeit.tex
  30 276 2489 diplomarbeit.txt
  ```

  Действие распространяется на большинство файлов:

  ```
  wc diplomarbeit_kapitel01.tex diplomarbeit_kapitel02.tex
  8 102 998 diplomarbeit_kapitel01.tex
  22 174 1491 diplomarbeit_kapitel02.tex
  30 276 2489 total
  ```

  Так, в системе Юникод один знак может занимать больше байтов, следовательно, показатель количества байтов гораздо менее информативен. Поэтому в новых версиях `wc` есть также возможность, которая позволяет отразить количество знаков (с помощью `-c` или `-m`).

  Кроме работы с текстовыми файлами `wc` подходит, например, для того, чтобы сравнить, содержат ли файлы или таблицы одинаковое количество элементов.

Искать и найти

Рассмотрим команды для поиска.

- `find`. Это разносторонняя команда, с помощью которой можно найти любые файлы, которые соответствуют определенным условиям. Она без требования просматривает даже те подкаталоги, которые не выполняют никаких Shell-команд. Например:

  ```
  find . -name "*hog.png" -print
  ```

  Точка означает «искать в актуальных каталогах», сюда же относятся подкаталоги. `-name` означает, что команда `find` рассматривает только имена файлов, а `"*hog.png"` указывает на то, что имя файла должно заканчиваться на `hog.png`, также будут найдены файлы `warthog.png` и `hedgehog.png`. `-print` означает лишь то, что команда должна передать вам результаты поиска. В большинстве новых систем можно обойтись и без `–print`. Попробуйте.

  ```
  find animals/hedgehogs -mtime 7 -print
  ```

  Эта команда найдет все файлы в каталоге `animals/hedgehogs`, которые были изменены в течение последних семи дней. Если речь идет о минутах, то вместо `-mtime` можете применить `–mmin`.

  ```
  find . -perm 777 -print
  ```

  Здесь команда находит все файлы, которые по праву доступа (см. ранее) обозначены как «доступно для всех»... что не является хорошей идеей. (Об этом подробнее говорится в главе 25.)

  Будьте внимательны: если даете команду `find / -name "*log"`, то можете смело идти пить кофе, потому что команда `find` просмотрит весь жесткий диск в поисках ключевого слова. `locate` — это однозначно более быстрая возможность поиска по большому дереву файлов.

- `locate`. Ищет все пути доступа, которые ключевое слово когда-либо включало в себя как подстрока.

  ```
  locate wallet.dat
  ```

  Здесь команда на всех дисках ищет файл, в имени которого содержится `wallet.dat`.

  `locate` выполняет практически ту же самую функцию, что и команда `find`, только гораздо быстрее, так как просматривает не весь жесткий диск, а лишь базу данных. Правда, для большинства Unix-систем необходимо сначала установить такую базу данных, что новички в пользовании командной строкой не делают ни в начале, ни в конце, когда команда не похожа на `locate --create` и называется `updatedb`. Если вам приходилось искать что-то с помощью команды `find` на больших дисках Unix-систем и из-за разочарования вы хотите перейти на систему Windows, советуем вам разобраться с `locate`.
- `grep`. Ищет в содержимом файлов и после небольшого предварительного просмотра места нахождения выдает найденные ячейки.

  ```
  grep "^<[Hh]1>" index.html
  ```

  В данном случае команда выдает список заголовков типа `H1`, которые встречаются в файлах `index.html`. Синтаксис такой же, как в стандартных выражениях: `^` означает «в начале строки», квадратные скобки следят за тем, чтобы в результате были найдены и `<h1>`, и `<H1>`. Результат поиска выглядит примерно так:

  ```
  index.html:<h1>любимый еж</h1>
  index.html:<H1>Противная пиявка</H1>
  ```

  Параметр `-i` (case-insensitive) позволяет проследить за тем, чтобы оба способа представления были найдены:

  ```
  grep -i "<h1>" index.html
  ```

  Параметр `-v` означает «найти все, что не противоречит условию».

  ```
  grep -v "index.html" access.log
  ```

  Данная запись выдает все строки с сервера лог-файлов, в которых кто-нибудь вызывал какие-нибудь другие страницы, кроме `index.html`.

  Параметр `-c` считает ссылки или источники. С помощью:

  ```
  grep -c приблизительно /диссертация/глава4.tex
  ```

  вы можете выяснить, как часто встречается слово «приблизительно» в главе 4 этой диссертации.

  ```
  grep -ir ungefähr .
  ```

  В записи выше команда просматривает каталог, в котором вы находитесь (`.`) и все подкаталоги (`-r` (recursive)), чтобы найти «приблизительно» и «Приблизительно» (`-i` (case-insensitive)).
- `which`. Иногда в системе существует несколько версий программы. Так, например, когда Python уже был готов к комплектации, была доукомплектована новейшая версия.

`which python` отвечает на вопрос «Какая из различных версий Python запустится в моей системе, если в команде я не укажу `which`?» в формате пути к этому файлу. Например, таким образом:

```
/usr/bin/python
```

Если не поступает никакого обратного сигнала, а вы точно знаете: то, что вам необходимо найти, находится где-то в системе, значит, команда зарегистрирована не в `PATH`. Система по очереди просмотрит список каталогов, чтобы найти там искомое. Затем выполняется то, что найдено первым. Использование `echo $PATH` позволяет отобразить содержание этого списка, с помощью двоеточия отдельные каталоги могут быть друг от друга отделены.

Содержание `PATH`-переменной также может быть изменено:

```
export PATH=$PATH:/new/path
```

Через двоеточие старое содержание `PATH`-переменной связано с новым путем, в данном случае `/new/path`. Если вы хотите удалить `PATH`-переменную из каталога, можете записать содержание во временный файл:

```
echo PATH=$PATH > tmp
```

После того как вы в любом редакторе по вашему выбору настроите содержание и сделаете исполняемым файл, который имеет ограниченное право доступа, с помощью `chmod +x tmp` вы сможете считать содержание, всего лишь вызвав данный файл:

```
./tmp
```

`$PATH` позволяет проконтролировать успешность операции.

Сбережение ресурсов

Здесь приведены команды для работы с ресурсами компьютера.

- `ps`. Команда `ps` (process status) позволяет отобразить список всех текущих процессов. Она особенно полезна для того, чтобы выяснить, не допущена ли где-нибудь нелепая ошибка, которая ведет к тому, что процесс часами идет по кругу по тому же `while`-циклу. Весьма кстати используются такие параметры, как `-A` («отображает все, даже процессы других пользователей») и `-f` («показать полную команду»).

  В списке, выданном с помощью команды `ps`, наиболее важными являются две вещи: в графах, подписанных `START` или `STIME`, видно, с какого времени идет процесс. Если процесс начался подозрительно давно или там указана просто дата, вероятно, он еще идет и его можно закончить с помощью команды `kill` (смотрите далее). Для этого необходим ID процесса, который можно найти в графе `PID`.
- `top`. Команда `top` (top CPU consumers) отображает информацию, схожую с отображаемой командой `ps`. Отображение актуализируется каждые две секунды и выглядит примерно так:

```
top – 15:28:45 up 478 days, 15:59, 1 user, load average: 0.61, 0.16, 0.08
Tasks: 181 total, 2 running, 179 sleeping, 0 stopped, 0 zombie
Cpu(s): 0.0%us, 4.7%sy, 50.8%ni, 44.4%id, 0.0%wa, 0.0%hi, 0.0%si, 0.1%st
Mem: 4190040k total, 2980332k used, 1209708k free, 345488k buffers
Swap: 4194296k total, 144912k used, 4049384k free, 2063132k cached
```

Здесь нас прежде всего интересует среднее значение загрузки системы (load average). Средняя загрузка отображается в виде трех значений, которые представляют собой усредненные величины загрузки процессора за последние 1, 5 и 15 минут. Что понимается под высокой или низкой загрузкой, зависит от соответствующей системы, а именно от количества CPU-ядер. Поэтому для того, чтобы иметь понятие о том, что представляет собой нормальное состояние компьютера, целесообразно вызывать команду `top`. Грубо говоря, загрузка процессора 1 и ниже является оптимальной, все, что ниже 3, считается нормальным. Это касается компьютеров только с одним CPU. С помощью:

```
grep -c processor /proc/cpuinfo
```

можно выяснить количество CPU в компьютере. Если их два, то оптимальной будет загрузка 2 и ниже, допустимо все, что ниже 6.

- `kill`.

```
kill 1234
```

С помощью этой команды дается указание закончить процесс под номером 1234. Бывают случаи, когда это не срабатывает и приходится действовать по-другому:

```
kill -912349 означает «Никаких возражений!». Если после этого наблюдаются беспорядочные остатки процесса, необходимо сначала попробовать вежливый способ.
```

- `du`. Неужели за день на сервер от других ничего больше не попадает? Вероятно, у вас бывали случаи, когда имеющееся пространство памяти (disk quota) было исчерпано. Команда `du` (disk usage) показывает, не занимают ли ваши данные слишком много места.

```
du –sh
```

Вместе с командой применяются параметры `s` (summary) и `h` (human-readable). Если вы запустите команду `du -sh /` на высшем уровне каталога, то узнаете, сколько места занято на диске. Результат будет не в виде длинных неудобных цифр без единиц измерения, а, наоборот, легко воспринимаемый, например, такой: 9.8G, где G означает «гигабайт». Также можно узнать, в каком каталоге находится самый большой «потребитель пространства», сделать это можно следующим образом:

```
du -sm * / | sort -n Используя параметр m, вы получите результат в мегабайтах, параметр human-readable здесь, к сожалению, не функционирует, чего требует следующая сортировка. Параметр -n выполняет числовую сортировку, для того чтобы на выходе было «1, 2, 10, 11» вместо «1, 10, 11, 2».
```

Совместная работа

Рассмотрим команды для совместной работы.

- `python -m SimpleHTTPServer 8080`. Запускает маленький веб-сервер через порт 8080[1], который передает файлы в каталог, в котором вы находитесь. Если ваш IP адрес 192.168.1.11, другие могут воспользоваться вашими файлами через http://192.168.1.11:8080/. Если вы хотите сделать доступными для всех пару текстовых файлов, фотографий из отпуска или что-то подобное, то вам будет достаточно данной команды. С ее помощью вы также можете позволить другим людям временный доступ к лог-файлам на сервере без предоставления дополнительного аккаунта пользователя. Бывают случаи, когда файлы `index.html` выдаются как главная страница наравне с «большими» веб-серверами, такими как Apache httpd. Если они отсутствуют в каталогах, разделенных Python, сервер выдает список файлов.
- `php -S 127.0.0.18080`. Так же, как описанная выше Python-команда, запускает веб-сервер, который через веб делает файлы доступными. Этот веб-сервер также выполняет PHP-сценарии.

Синхронизация

Рассмотрим команды для синхронизации.

- `cron`. Команду `cron` можно представить как таймер. Вместо того чтобы поливать балконные растения или включать отопление, эта команда запускает сценарии на сервере. Она очень полезна, так как человеку не придется больше думать, например, об открытии резервных копий или регулярном удалении ненужных файлов, которыми перегружен сервер. Команда `cron` действует только в том случае, когда определенные процессы можно откорректировать по времени: возможно, вам нужны данные небольших веб-средств из другого источника, например, когда вы хотите на боковой панели своего сайта показать свои последние сообщения в Twitter. Во-первых, если вы при каждом вызове сайта контактируете сначала с чужим сервером, то у нетерпеливого посетителя это займет больше времени, чем если бы данные находились на самой странице. Во-вторых, это вредит другим серверам и злит их администраторов, особенно если ваш сайт вызывают чаще нескольких раз в день. И в-третьих, это абсолютно не нужно, потому что зачастую необходимы только самые актуальные данные (секундной актуальности). Как правило, вполне достаточно просмотреть то, что было изменено, а затем сохранить эти данные локально и использовать дальше.

  При программировании таймера придется также повозиться с запутанными переключателями. Но в Сети довольно много подробных примеров. Каждый

---

[1] Обычно веб-сервер применяет порт 80, но для более быстрого обмена файлами используется, как правило, 8080, только процессы с компьютера администратора требуют получения доступа от порта меньше 1000.

запрос находится в файле под именем `crontab` в своей собственной строке, выглядит это примерно так:

```
051 * * * nice -n 19 php /home/bernd/backup.php >> /home/bernd/cronjob.
log 2>&1
```

Первые пять позиций, из которых три здесь заменены звездочками, указывают на промежуток времени, в течение которого что-то должно произойти. Первая позиция указывает на минуты, вторая — на часы, затем следуют дата, месяц и день недели. Звездочки означают любое время. Согласно тому, что здесь написано, каждый день каждого месяца в 1.05 будет вызываться PHP-сценарий с именем `backup.php`. `nice -n 19`. Это означает «дорогой сценарий, будь так любезен и не подвергай сервер чрезмерным нагрузкам. Если ему одновременно нужно выполнить еще 18 заданий, будь первоочередным». И когда резервный сценарий сообщит вам либо об успешном выполнении, либо об ошибке, в конце должен быть прикреплен файл с именем `cronjob.log`. Через `2>&1` сообщение об ошибке, как и данные вывода сценария, будут записаны в лог-файл.

Чем дальше, тем очевидней. Проблема заключается в том, что `crontab`-файлы непросто локально изменить в любом редакторе по вашему выбору, так как они находятся внутри системы. Для этого вам придется проинформировать `cron` об изменениях. Проще всего сделать это, если для начала вы сохраните содержание `crontab` во временных файлах:

```
crontab -l > cronjobs.txt
```

Можете обрабатывать эти файлы либо на сервере, либо локально через FTP-Client. С помощью `crontab cronjobs.txt` вы информируете `cron` о новых Cron-заданиях.

Если вместе с `crontab` вы используете параметр `-l`, то будут показаны новые задания.

Редактирование с сервера

Как правило, в системе Unix вы можете получить доступ как к одному, так и к нескольким редакторам. Какими они будут, зависит от дистрибутива. Если вы не знаете, что это дает, попробуйте `which vi` или `which nano`.

- vi. Это редактор, к которому необходимо привыкнуть. Вам следует начинать его использовать только после того, как прочтете хотя бы основополагающие инструкции. Для непосвященных vi довольно прост: в нем всего два режима, в одном он жалуется, а в другом портит текст. Если понять, что придется работать с Esc и :, то можно действительно продуктивно редактировать тексты. vim — новая улучшенная редакция. На сайте vim-adventures.com в игровой форме можно научиться пользоваться vim.
- nano. Более дружелюбный по отношению к новичкам редактор, который показывает имеющиеся команды в двух строках. Самой важной командой является Ctrl+G, которая вызывает помощь.

Интернет

Рассмотрим средства для работы в Сети.

- curl. Программа curl — как маленький швейцарский перочинный нож для получения доступа к веб-серверу. Чтобы открыть веб-страницу в текстовом редакторе, можно банально загрузить ее на локальный диск. Из этого следует, что таким образом можно, например, каждый день синхронизированно скачивать файл с биржевыми курсами, анализировать его содержание и записывать в базу данных.

  Гораздо больший интерес curl представляет для веб-разработчиков, так как может использоваться для целенаправленных HTTP-запросов. Например, для сайта AJAX вы написали веб-сервис, который передает свои данные как JSON, и хотите этот веб-сервис протестировать, в таком случае вам лучше использовать curl, чем обычный браузер. Если вы откроете URL веб-сервиса через браузер, он ответит, что поддерживает форматы HTML, XML или обычный текст, но не JSON — формат вашего веб-сервиса. Ваш веб-сервис и браузер не смогут найти общий формат, и вы увидите лишь печальную ошибку страницы.

  `curl -H "Accept: application/json" http://example.com/webservice -o webservice.json`, наоборот, вызывает URL и запрашивает содержание как JSON — именно так, как передает ваш веб-сервис. Файлы `webservice.json` (по умолчанию пишите `curl` после `stdout`, `-o` выдает файлы как цель) вы можете либо открыть с помощью текстового редактора, либо просмотреть с помощью `grep`.

  curl может также имитировать загрузки, например, с помощью флажка `-X` вы можете установить `POST` как HTTP-глагол (по умолчанию это глагол `GET`). Если вы вызываете программы с помощью флажка `-i`, он выдает вам HTTP-заголовок, который отправляет веб-сервер. С помощью `-v` вы можете прочесть итоговый обмен данными между curl и веб-сервером от установления TCP-соединения, до ликвидации, после того как сервер ответит. Вам не обязательно это понадобится, но если вы когда-нибудь упорно боролись с проблемой обмена данными с сервером, тогда вы оцените помощь curl.

  Конечно, существуют еще и расширения браузера, которые позволяют изменять заголовки запроса, но вы можете проверить curl-строки просто как тест с помощью кода вашего веб-сервиса в системе контроля версий. Можете запустить curl на сервере, на котором в вашем распоряжении нет графического пользовательского интерфейса, и вызывать ее из сценария. Последний пункт особенно интересен для программистов, так как с помощью curl-вызова возможно заменить большое количество кодов. libcurl — это библиотека, в которой размещены функции для всех имеющихся программ, например для PHP.

- ssh. ssh (Secure Shell) — это библиотека средств, необходимая для того, чтобы установить зашифрованную и поэтому более безопасную связь компьютера с кем-нибудь еще. Если у вас есть сервер с доменом `example.com`, тогда вы можете установить связь между сервером и терминалом вашего локального компьютера с помощью `ssh username@example.com`. При этом `username` — ваше имя пользователя на сервере.

Например, если человек хочет работать на Linux-сервере, ему придется войти в систему не так, как в Windows, через удаленный Рабочий стол, а работать преимущественно с командной строкой сервера. ssh шифрует передаваемые данные в большинстве случаев так надежно, что вы можете войти в систему своего сервера через открытую беспроводную локальную сеть (WLAN), не опасаясь, что другие пользователи этой сети смогут прочитать ваш пароль. Ни в коем случае вы не должны входить в систему через WLAN незашифрованно, потому что так вы даете другим возможность записать радиосвязь и подобрать пароли.

- scp. Если вы в основном работали с FTP, вам, безусловно, необходимо перейти на scp или sftp, так как FTP не зашифрован: если вы пользуетесь открытой беспроводной локальной сетью, каждый может без особых затруднений читать ваши данные вместе с вами. Эта часть ssh-пакета позволяет производить зашифрованный обмен данными через сеть между одним компьютером и другими. При этом вы можете скачивать и загружать файлы на сервер, и никто, за исключением секретных служб, не сможет их прочитать. Загрузка происходит так:

  ```
  scp datei.txt username@192.168.1.11:/data/
  ```

  Файл `datei.txt` загружен с компьютера, IP-номер которого `192.168.1.11`, в текущий каталог `/data/`. Скачивание происходит так:

  ```
  scp username@192.168.1.11:/data/datei.txt ./
  ```

Стоит ли помнить обо всем этом?

Нет. Одной из положительных и очень полезных характеристик оболочки Unix является то, что она предоставляет журнал команд, которые вы набрали на клавиатуре. Этот журнал избавляет вас от повторного набора. Для того чтобы работать с этим журналом и получить доступ к записям, есть несколько путей.

- Клавиши со стрелками вверх/вниз. При нажатии клавиши со стрелкой вверх будет отображена последняя команда, использовавшаяся в оболочке. При нажатии клавиши возврата вы вернетесь в прежнее место. При повторном нажатии клавиши со стрелкой вверх появится предыдущая команда и т. д. Для того чтобы вернуться назад к новым командам, воспользуйтесь клавишей со стрелкой вниз. Нажимайте клавишу до тех пор, пока не появится пустая строка ввода.
- Ctrl+R. Для того чтобы целенаправленно найти определенную команду, воспользуйтесь клавишами Ctrl и R. Правда, вы можете получить только часть той команды, которую ищете, и оболочка отразит лишь первую обнаруженную команду. Например, если вы вошли в систему сервера через:

  ```
  ssh -l johannes bayeux.datensalat.net
  ```

  то можете эту строку как по мановению волшебной полочки найти с помощью Ctrl+R или набрав на клавиатуре `bayeux`. Будьте внимательны: когда вы нажимаете клавишу ввода (Enter), найденная команда сразу же выполняется.

- `history`. Это команда, которая выдает ваш журнал истории. В нем вы увидите последние пару сотен команд, которые вводили. У каждой команды есть свой номер в журнале истории, выглядит это примерно так:

```
14 cd /usr/local/tomcat/logs/
15 tail -n 100 -f catalina.out
16 tail -f tomcat.log
```

  Если хотите повторить команду, можете поставить перед номером команды восклицательный знак, например, `!15` для первой команды `tail`. `!!` — это последняя команда, и ей соответствует клавиша со стрелкой вверх. В отличие от Ctrl+R и клавиш со стрелками, `!номер` не помещает найденную команду в строку ввода. Но если вы нажмете клавишу Esc, команда все равно будет выполнена.

Еще не всё!

В этой главе мы разобрали не все тонкости bash-программирования, но тем не менее дали пару советов, которые сделают возможным выживание с командной строкой и помогут упростить трудовые будни программистов. Вероятно, вы скажете о том, что в Интернете данное проблемное поле в целом лучше освещено и документировано, но не стоит забывать о том, что эти средства являются «бульоном», из которого впоследствии был приготовлен Интернет. Вы без проблем найдете помощь на форумах или отыщете параметры отдельных инструментов в Сети.

Если вы по профессиональным причинам основательно работаете с командной строкой или этот раздел пробудил ваше любопытство, можете подыскать себе что-нибудь на сайте www.commandlinefu.com. Сайт предлагает много небольших примеров полезного использования команд командной строки, которые вы можете отыскать либо просмотреть по критерию «оценить особенно полезные». Для этих фрагментов обычно любезно дан небольшой комментарий. Unix Power Tools[1] — это книга, несмотря на звучный заголовок, рассчитанная на профессиональную аудиторию и полезная для новичков, так как в ней есть подробное объяснение всех команд.

---

[1] Книга Unix Power Tools вышла в издательстве O'Reilly в 1997 году и с тех пор была несколько раз переработана. Будьте внимательны — пользуйтесь последним изданием.

23 Объектно-ориентированное программирование

> Студенты смотрят на ворох информации, который я им представляю, и говорят: «Ах, ведь можно записать все это и последовательно, тогда будет понятнее».
>
> *Роланд Краузе, биоинформатик*

Как правило, многие люди хотят в жизни чуть больше порядка. Хорошо, когда есть много карандашей, деталей от «Лего» и записных книжек, но было бы еще лучше, если бы ночью не приходилось натыкаться на эти повсюду разбросанные вещи. И потому необходимость в решении проблемы хранения довольно велика: если запихать все в одно место, то не только не придется наступать на свои вещи, но и можно будет примерно представлять, в каком ящике они находятся.

Однако у метода собирания всего в коробочки есть свои недостатки, потому что в какой-то момент вы уже не знаете, что в какой ящик убрали. В таком случае нужно упаковывать вещи по принципу подобия: ручки в одном ящике, а детские игрушки — в другом, специальном ящике для игрушек (или в пяти ящиках). Тогда, возможно, вы все еще не будете точно знать, где стоит коробка с USB-кабелем, но вам придется искать уже просто коробку, а не отдельную вещь, а это, как показывает опыт, гораздо проще.

Зачастую процедурное программирование и похоже просто на хранение всего в одной коробке: разнообразные функции и переменные помещаются в один файл, но если вы ищете конкретную функцию, то без хорошей памяти тут не обойтись. И хотя поиск по файлу помогает, но, если вы только примерно знаете, как называется функция, и он не особо много дает. Основная проблема заключается в том, что не существует никаких технических мер, которые бы вынуждали программиста придерживаться какого-либо определенного порядка.

А объектно-ориентированное программирование (ООП), напротив, позволяет организовывать исходный код в соответствии со вторым методом. Все, что относится к объекту, например переменные и функции, записывается в один файл, в котором можно найти исключительно то, что имеет отношение к этому объекту. Если отступить от этого правила, то компилятор будет ругаться.

Начинающим программистам сложно сделать выбор между объектно-ориентированным и процедурным подходом при написании программы. Они, вероятно, читали, что объектно-ориентированный метод — современный и популярный, но им не до конца понятно, что конкретно этот метод должен привнести в разработку программного обеспечения. Эта неопределенность возникает частично из-за того, что в объектно-ориентированном программировании многое называется совершенно иначе, чем в процедурном программировании, даже если работает так же. Кроме того, объектно-ориентированное программирование зачастую позиционируется как принципиально отличное от процедурного. Не слишком беспокойтесь из-за этого, потому что, хотя с помощью ООП вы и изучаете другой мысленный подход к решению проблемы, это не значит, что вы с самого начала должны думать иначе.

На самом деле исторически объектно-ориентированное программирование является развитием процедурного программирования. Оба этих способа одинаково эффективны, так что нет такой проблемы, которую нельзя было бы решить и процедурно тоже. Объектно-ориентированное программирование просто более ясное и структурированное. Это звучит несколько категорично, но погодите немного, сейчас мы перейдем к обоснованию.

Объектная ориентированность не предполагает того, что вы теперь должны выбросить из головы все свои чудесные знания о процедурном программировании, переменных и функциях. То, что вы знаете из процедурного программирования как функцию, называется в ООП методом, и он состоит из привычных процедур. Особенностью ООП является то, что методы вместе с переменными (которые здесь предпочитают называть членами) компонуются в специальные модули, которые потом называются объектами.

Для программиста разница между процедурным и объектно-ориентированным мышлением является по большей части вопросом перспективы.

Это хорошо можно объяснить на примере кулинарных рецептов.

Рецепт описывает, как взвешивать ингредиенты, смешивать и нагревать. Все делается последовательно во времени, по крайней мере в хороших рецептах. Сначала надо взвесить ингредиент А, потом ингредиент Б, затем смешать ингредиенты А и Б, потом жарить и т. д. Сами ингредиенты являются пассивными компонентами, которые повар обрабатывает каким-то определенным образом. Шаги к конечной цели описаны в мельчайших подробностях.

Мысленная позиция объектно-ориентированного программиста скорее похожа на роль, которую шеф-повар играет на кухне ресторана: он знает рецепт и может делегировать его части. Шеф-повар понимает, что он может дать ассистенту достаточно знаний об отварном картофеле, чтобы затем дать тому задание исключительно по готовке отварного картофеля, не объясняя каждый шаг. Каким образом помощник сварит картофель, это его дело до тех пор, пока выходит нужный результат.

Таким образом, в этом упрощенном примере есть разделение труда: помощник повара готовит свою картошку наиболее эффективным для себя способом, а шеф-повар проверяет только результат и в остальном заботится о тех подготовительных шагах, которые входят в его собственную зону ответственности. Это ограждает его от микроменеджмента и разгружает от мелочей.

Преимущества объектно-ориентированного программирования

Исторически объектно-ориентированное программирование возникло в результате кризиса. В конце 1970-х емкость хранилищ компьютеров сильно увеличилась за счет перехода на интегральные схемы. Разработчики ПО того времени задействовали новые ресурсы, чтобы писать более мощные и удобные в использовании программы, что означало больший размер исходного кода для каждого проекта. В 1972 году голландский информатик Эдсгер Дейкстра следующим образом описал это явление в статье о ранней истории своей профессии.

> Основная причина кризиса программного обеспечения — резкий рост мощностей вычислительных машин. Проще говоря: нет вычислительной техники — нет проблем с разработкой программного обеспечения для нее; когда у нас было только парочку слабых компьютеров, в программировании не было больших проблем, сейчас же, когда у нас есть гигантские компьютеры, программирование стало столь же гигантской проблемой.
>
> *Эдсгер В. Дейкстра. Смиренный программист*

И то, что для него было гигантским компьютером, ничто по сравнению с начинкой современных смартфонов.

Из-за того что каждая функция могла быть вызвана из любого места в программе, было очень тяжело менять код, применяя рефакторинг (см. главу 15), потому что каждое изменение функции могло повлечь за собой изменения в тех местах, из которых эта функция вызывалась.

Кроме того, проекты расширялись слишком быстро для того, чтобы один-единственный программист мог сохранять полный контроль над всем.

Таким образом, задачи необходимо было разделить на модули и за каждый модуль отвечала своя команда. Чтобы после получить функционирующую программу, модули нужно было соединять с помощью интерфейсов. Вскоре исходный код еще бóльших проектов уже не получалось контролировать с помощью имеющихся средств. Разработчики потеряли контроль над ситуацией и стали реализовывать функциональность по нескольку раз, потому что не знали, что она уже была реализована до этого. В этой ситуации ООП действительно помогло, потому что обеспечило беспрепятственное взаимодействие многих программистов в рамках работы над большими проектами, а также облегчило объявление интерфейсов между частями программы.

В особенности усугубило этот кризис появление графических пользовательских интерфейсов (GUI) в 1980-х годах, потому что их элементы, например такие, как окна или меню, предусматривали необычайно насыщенные и при этом сложные взаимодействия. В то же время элементы графического интерфейса особенно удобно программировать с помощью объектно-ориентированных средств, что сделало эту, тогда еще новую, технологию интересной для разработчиков. Кнопка включения в GUI, например, имеет несколько состояний: обычное, выбранное и неактивное.

В зависимости от состояния такие элементы интерфейса могут принимать другой внешний вид, что требует относительно большого внутреннего кода и нескольких переменных, притом что программа, в которой данная кнопка просто будет использоваться, не должна знать об этом коде. Поэтому эти кнопки удобно использовать в качестве многоразового модуля: как разработчик программ с GUI, я должен просто запомнить, что у кнопки переключения есть различные состояния и что нажатием можно вызывать заданную мною функцию. Меня может не волновать, как именно это происходит, и тем самым я уберегаю себя от целого ряда сложностей.

Объектно-ориентированное программирование полезно использовать по многим причинам.

Объекты содержат указания на то, как их нужно использовать

Поскольку методы объекта принадлежат этому объекту, то по ним можно понять, что с ним можно делать.

Таким образом, объект определяет интерфейс по отношению к внешнему миру. В процедурном программировании функции и данные размещаются рядом без какой-либо связи, поэтому не так просто понять, какие данные можно обрабатывать с помощью каких функций.

Код более надежен, поскольку объекты автономны

Код, принадлежащий объекту, виден для всего мира только частично. В нем могут содержаться закрытые (приватные) методы, доступ к которым можно получить только через объект. Это повышает удобство сопровождения, потому что не все изменения влекут за собой изменения в удаленных частях кода (у локальных изменений локальный эффект). В хорошо написанном процедурном коде также уделяется большое внимание надежности, но ООП еще и поддерживает разработчиков в данном вопросе. Постоянно писать модульный код с помощью языка без встроенной поддержки удается немногим — это не всегда легко и с ООП.

Стандартизация

ООП — это метод организации и структурирования кода, который не приходится утомительно объяснять людям, с которыми хотите работать. Есть и другие методы, но желательно иметь некоторый общий стандарт — и ООП является как раз таким широко распространенным на сегодняшний день стандартом.

Все, что связано, группируется вместе

Благодаря тому что переменные и код, который использует эти переменные, объединены в одном объекте, код становится более понятным, поскольку легко можно проследить взаимосвязь. У каждого объекта есть имя, которое, как в нашем примере с игрушками, указывает на содержащиеся в нем функции.

В процедурном программировании было бы очевидным, например, поддержка в базе данных функции `getUserId()`, а в другом файле в системе управления пользователями была бы функция `deactivateUser()`. В объектно-ориентированной системе был бы создан объект пользователя, а в нем — оба метода: `User-> getId()` и `User->deactivate()`.

Сложность инкапсулируется

Сложность кода часто заключается в избытке деталей и условий, которые необходимо соблюдать. Эта сложность не исчезает с ООП, но она инкапсулируется в объекте с помощью упомянутых ранее приватных методов. (Далее принцип инкапсуляции будет объяснен более подробно.) Объект может содержать еще 100 методов, но если вы просто хотите использовать его, то вам потребуется всего лишь несколько методов, которые видны снаружи класса. И вам не придется разбираться со сложной структурой объекта полностью.

ООП упрощает процесс моделирования данных

В ООП всегда делаются попытки найти сходство между родственными данными или частями кода, чтобы затем перейти от этих конкретных родственных примеров к общему случаю, который включает конкретные случаи. И хотя специалисты, использующие процедурное программирование, также стремятся получить абстракцию путем перехода от конкретных случаев к общим правилам, но в ООП поиск общего решения особенно эффективен за счет использования механизма наследования. Наследование позволяет изменять некоторые свойства родительского объекта в конкретных случаях. Детали будут приведены в дальнейшем.

Принципы объектно-ориентированного программирования

Модульность и инкапсуляция

Если программное обеспечение разрабатывается в течение длительного времени, то часто наступает такой момент, когда каждая часть программы может вызывать любую другую функцию. Если это кажется вам знакомым, то вы, вероятно, знаете и о том, что такое положение вещей не дает сделать все быстро в случае, если когда-нибудь понадобится настроить какую-либо функцию. Структура системы очень плотная — разные части программы переплетены друг с другом. Такие тесно сцепленные системы содержат код, который лишь с трудом можно отделить и повторно использовать в других системах. И довольно часто изменение в одном месте влечет за собой дальнейшие изменения в совершенно других областях кода.

Если вы хотите избежать такой спаянной кодовой массы, то должны создать закрытые области, которые напрямую доступны только из некоторой части исходного кода, в то время как другие области могут быть доступны только через интерфейсы. Тогда вы будете знать заранее, какие части кода затронут изменения, потому что это могут быть только функции из одного и того же объекта, поскольку они «живут» в одном и том же «черном ящике».

К примеру, вы разрабатываете систему управления контентом, в которой есть как пользователи с такими простыми правами, как, например, «создать статью» или «читать статью», так и администраторы с расширенными правами. Тогда вам необходимо

иметь возможность постоянно запрашивать уровень авторизации каждого пользователя, чтобы включать или блокировать функции в пользовательском интерфейсе.

В нашем примере создавать статьи могут все администраторы и постоянные пользователи, доступ которым был бы дан явно.

В случае, если бы вы хранили пользователей в массиве:

```
var $users = [
    { id: 1, name="Hans Meiser", mayWrite = true, level="user"},
    { id: 2, name="Bernd Lauert", mayWrite = false, level="admin"},
];
```

Везде, где пользователь может создавать новые записи, проверьте следующее.

- Если пользователь админ, то он может создавать записи независимо от того, `mayWrite` имеет значение `true` или `false`.
- Если пользователь простой юзер, то он может создавать записи, только если `mayWrite` имеет значение `true`.

Конечно, вы можете внести это условие в функцию, но перед тем, как ее вызывать, вам все равно придется выбрать нужного пользователя из `$users` и передать его этой функции на проверку. Переменная `$users` также может быть изменена из-за ошибок в других областях кода, возможно, даже в тех, которые не имеют ничего общего с написанием статей.

При объектно-ориентированном подходе вы бы создали объект, который может сделать проверку и в то же время содержит информацию о пользователе:

```
object User = {
    private var $id;
    private var $name;
    private var $mayWrite;
    private var $level;

    function hasWritePrivileges() {
        if ($level == "admin") {
            return true;
        }
        return $mayWrite;
    }

    function getId() {
        return $id;
    }

    function getName() {
        return $name;
    }
}
```

Функция, проверяющая, разрешено ли пользователю создавать статьи, включена в данном случае в объект. В нашем выдуманном ООП-синтаксисе эта функция видна извне, в то время как поля `$mayWrite` и `$level` скрыты. Условия того, может

ли пользователь на основе разрешения администратора или своего уровня писать статьи, проверяются только в объекте, в методе `hasWritePrivileges()`. Существенным моментом здесь является то, что функциям, разрешающим или запрещающим определенные действия в пользовательском интерфейсе, не нужно ничего знать о правилах управления правами — они предусмотрены в объекте пользователя. Ни уровень пользователя, ни его разрешение не могут быть изменены извне вследствие программной ошибки, они устанавливаются при создании объекта.

Этот принцип инкапсуляции (от англ. encspsulation) является важным принципом ООП. Он уменьшает сложность программы, в то время как внутренний принцип работы объекта остается нетронутым. И в данном случае понятие «объектно-ориентированный» снова во многом соответствует действительности: и в реальном мире мы, как правило, не знаем точно ни на техническом уровне, ни на уровне программного обеспечения, как работает компьютер. Все эти сложности упакованы в симпатичную обертку, а мы лишь используем систему в целом.

Каким образом `hasWritePrivileges()` определяет права, снаружи неясно. Если вы когда-нибудь захотите включить управление правами в базу данных, то сможете сделать это путем изменения `hasWritePrivileges()`, не трогая остальную часть программы. А если вы когда-нибудь будете вводить новый уровень авторизации, например «Редактор», то вам придется изменить логику `hasWritePrivileges()`, а также лишь частично изменить переменные типа `$level`. Внешне ничего не поменяется, вы будете и дальше делать проверку, вызывая `hasWritePrivileges()`.

Объекты различают изнутри и снаружи. Для методов объекта пользователя поля (переменные) `$mayWrite` и `$level` — внутренние. Функция `hasWritePrivileges()` также внутренняя, и поэтому она может получать доступ к значениям полей. Снаружи эти значения недоступны, поскольку они заявлены как приватные[1]. Таким образом, следующий код был бы ошибочным:

```
var $user = new User();
print ($user->$level);
```

Команда `print` пытается извне получить доступ к закрытой переменной `$level`, что недопустимо. Смысл этого ограничения в том, что если внешний мир хочет что-либо узнать о переменных, которые заявлены как частные, то он должен смиренно запрашивать эту информацию у объекта. Подобно тому как локальные переменные функции не могут быть изменены или прочитаны за пределами этой функции, так и закрытые переменные являются локальными для своего объекта.

Абстракция данных

Объектно-ориентированное программирование предназначено для того, чтобы помочь программисту в выведении абстракции путем перехода от похожих проблем к общей сути и различным деталям.

---

[1] Существуют также объектно-ориентированные языки без закрытых переменных, например Python. В таких случаях контроль за тем, чтобы ничего не натворить с внутренней работой объекта, входит в обязанности программиста.

Например, для того, чтобы иметь возможность отображать в графических редакторах различные форматы изображений, в процедурном программировании пришлось бы написать множество функций:

```
function renderJpeg ($x, $y, $width, $height, $data){
    ...
}

function renderPng ($x, $y, $width, $height, $data){
    ...
}

function renderTiff ($x, $y, $width, $height, $data){
    ...
}
```

Не нужно быть высококлассным программистом, чтобы понимать, что между форматами растровых изображений наряду с некоторыми различиями есть и целый ряд сходных черт: изображения состоят из пикселов, у них есть высота и ширина, и они могут быть отображены в любом месте на экране. Это описание — абстракция, потому что мы не учитываем, какой из форматов изображения обладает прозрачностью, изображение в каком формате можно сжимать без потерь и закодированы ли цвета с использованием 8 бит (это 256 цветов, например GIF) или 24 бит (16,7 млн цветов с прозрачностью, например, JPEG).

Было бы хорошо, если бы у нас была только одна `renderImage`-функция. Но в процедурном программировании приходится либо писать для каждого графического формата собственную функцию, либо исследовать содержимое функции на предмет того, какой графический формат заключен в `$data`, чтобы затем вызывать специфические для формата функции декомпрессии, что привело бы к ужасающе длинным и сложным функциям.

В ООП гораздо легче вывести эту абстракцию посредством определения Image-объекта:

```
object Image = {
    var $height;
    var $width;
    var $data;

    abstract function render ($x, $y);
}
```

Помимо переменных `$height`, `$width` и `$data`, есть еще своего рода шаблон для `render()`-функции. Он не реализован (это задача разработчиков, которые хотят написать специфический код для определенных графических форматов) и, следовательно, определяется как абстрактный. И здесь мы устанавливаем, что каждая конкретная `render()`-реализация получает две переменные: `$x` и `$y`, отвечающие за место на экране, на которое будет выведено изображение.

Используя упомянутый ранее принцип наследования, теперь можно определять специфические объекты для представления различных графических форматов:

```
object PngImage inherits Image {
    function render ($x, $y) {
        ...
    }
}
```

Этот обратный шаг — от абстракции к конкретным процессам реализации — мы опишем в подразделе «Полиморфизм» данного раздела.

Из-за того что код ограничен подобием разных форматов изображений, он в чем-то хуже адаптирован под особенности каждого формата. Формат, поддерживающий 8-битный режим, лучше подходит для тех случаев, когда в мониторе использована тоже только 8-битная матрица. Но так как мы хотим поддержать более качественные форматы, то функция `render()` 8-битного формата должна отталкиваться от того, что глубина цвета монитора составляет 24 бита, и в соответствии с этим должна перекодировать данные.

То, что на первый взгляд кажется недостатком, имеет на практике значительные преимущества, поскольку сведение похожих типов до их сходных черт значительно упрощает использование кода. Упрощает, потому что позволяет абстрагироваться от малосущественных деталей. Как разработчику мне не нужно знать, для каких особых оптимизаций или возможностей особенно подходит формат X, вместо этого я могу запомнить простой, охватывающий многие форматы способ получения изображения на экране. За то, как именно это происходит, ответственен тот, кто разрабатывает программную поддержку конкретного формата.

Абстракция выделяет сущность, ядро группы связанных структур данных или блоков кода и убирает специфические детали.

Абстракция создает четкое разделение между теми программистами, которые хотят просто использовать модуль кода (в нашем случае для создания изображения), и теми, которые хотят написать новые модули кода (в нашем случае для поддержки новых форматов изображений). Первым для решения задачи достаточно правильно вызвать функцию, в то время как вторые должны лишь писать функции. Но и тем и другим незачем упоминать о самом интерфейсе — он уже определен с помощью абстрактного объекта.

Для того чтобы достичь успеха в ООП, рекомендуется заранее привыкать к поиску свойств структур данных или программных компонентов, которые похожи друг на друга до такой степени, что они могут быть абстрагированы. Так, в ERP-системе имеет смысл свести все продукты к одному объекту, который обладает такими свойствами, как цена или сроки доставки. В системе обработки изображений базовым объектом был бы фильтр. Каждая конкретная реализация объекта фильтра реализует свой метод, который изменяет входное изображение. Как этот метод работает, зависит от фактической реализации, базовый объект не имеет с этим ничего общего.

Если вы сначала определите свои абстрактные базовые объекты и их свойства, а уже затем запрограммируете конкретную реализацию, то логически поделить код хорошо получится само собой.

Полиморфизм

После того как вы абстрагировали общие свойства объекта, вам необходимо позаботиться о различиях, чтобы иметь возможность использовать этот код в различных конкретных случаях. В приведенном ранее примере мы описали с помощью объекта Image основные функции изображения. И теперь для того, чтобы целенаправленно что-то начать делать с кодом, необходимо разработать специфичные объекты Image для разных форматов изображений:

```
object PngImage inherits Image {
    function render ($x, $y) {
        ...
    }
}

object JpegImage inherits Image {
    function render ($x, $y) {
        ...
    }
}
```

Бывшая до этого абстрактной функция `render()` теперь будет оживлена с помощью кода, который может распаковывать (из `$data`) закодированные в PNG или JPEG данные и отрисовывать пикселы на экране. Переменные `$height`, `$width` и `$data` являются частью родительского класса, и, следовательно, их не нужно еще раз определять для специфических объектов. Координаты `$x` и `$y`, в которых должно быть показано изображение, не являются частью объекта, потому что, возможно, это же изображение захочется показать и в других частях экрана.

Только благодаря полиморфизму с помощью ООП можно создать действительно модульный и повторно используемый софт, потому что разрабатываемые интерфейсы остаются неизменными для множества различных, но при этом связанных областей. Неважно, какой графический формат мы хотим поддержать, — основополагающие функции остаются прежними. Мы можем быть уверены в том, что все графические форматы поддерживают эту функцию, и здесь нас не интересует, каким образом это достигается.

Взаимодействие абстракции и полиморфизма облегчает повторное использование кода, потому что его можно вставлять всегда одним и тем же образом в различные среды. Так, в нашем примере было бы очень просто осуществить интеграцию нового графического формата, тем не менее, возможно, сам процесс написания соответствующей функции `render()` для этого формата был бы довольно трудоемким.

Наследование

В объектно-ориентированном программировании наследование играет важную роль. К сожалению, и этот термин лишь частично соответствует тому, что мы обычно понимаем под наследованием. В то время как в реальном мире характеристики

родителей только раз наследуются детьми, а затем те ведут совершенно отдельную жизнь, объекты остаются связанными: если изменяются свойства родительского класса, то меняются и свойства производных классов. Общим является то, что и тут и там свойства наследуются и нет необходимости создавать их заново.

Наши ранее введенные классы `PngImage` и `JpegImage` связаны друг с другом, они потомки класса `Image`. В то время как код, с помощью которого распаковываются PNG- или JPEG-файлы, меняет свой внешний вид каждый раз в зависимости от графического формата, у дочерних объектов есть общие черты. Например, если потребуется узнать ширину или высоту изображения в пикселах, без разницы для какого формата, то, конечно, вы не захотите для этого заново переписывать код для каждого формата изображения.

Здесь в игру вступает наследование: вы пишете код только один раз как метод в объекте `Image`, и его потомки наследуют этот метод. Это выглядит следующим образом:

```
object Image = {
    var $height;
    var $width;
    var $data;

    abstract function render ($x, $y);

    function getWidth () {
        return $width;
    }

    function getHeight () {
        return $height;
    }
}

object PngImage inherits Image {

    function render ($x, $y) {
        ...
    }
}
```

Итак, чтобы создать объект `PngImage`, нужно написать:

```
var $png = new PngImage(url);
print ("Breite: "+$png.getWidth()+" Höhe: "+$png.getHeight());
```

Как вы видите, в определении объекта `PNGImage` ничего не изменилось, но он получил функциональность, потому что мы реализовали методы `getWidth()` и `getHeight()` в родительском классе. И в тот же момент все другие специфические форматы изображений эту функциональность... ну-у-у, унаследовали. Вместо того чтобы расширять какой-то один определенный формат изображения, мы разом сделали все форматы умнее и удобнее.

Конечно, это не чудодейственное средство. В процедурном программировании вы просто продублировали бы код для различных функций изображений, один код для высоты, другой — для ширины. Тем не менее это потом необходимо было бы проделать и для всех других форматов.

Включение методов в родительский объект имеет три эффекта.

1. Если программист видит код, то он знает, что методы `getHeight()` и `getWidth()` могут быть вызваны для всех графических форматов. Если бы эти методы были включены в объекты `PngImage` и `JpegImage`, то мог бы существовать третий формат изображения, у которого этих методов не было бы.
2. Каждый формат, который мы хотим поддержать, должен обладать свойствами высоты и ширины. Таким образом явно будет показано одно условие, которое с легкостью можно рассматривать как известное, не проверяя его.
3. Производные классы содержат только специфический код для своего графического формата. Родительский класс содержит только более общий код. Это позволяет упорядочить объекты и четко разделить задания. В большинстве случаев вы только будете смотреть, какие методы содержит родительский класс, потому что для отображения изображений не обязательно учитывать специфику графических форматов. Таким образом, наследование поддерживает полиморфизм.

Разумное использование ООП

ООП всегда хорошо использовать тогда, когда вы имеете дело с данными, между которыми может происходить много взаимодействий. Подумайте, к примеру, о системе интернет-магазина. Требования к такой системе следующие.

- Клиенты могут выбирать товары (товарные позиции) и добавлять их в корзину.
- Клиенты могут видеть общую стоимость.
- Клиенты могут заказать товары из корзины или очистить корзину.

Если вы захотите перенести эти требования в программное обеспечение, то сразу увидите, что их можно легко реализовать с помощью различных логических единиц, таких как склад, содержащий товары, клиент, корзина и различные товарные позиции. Эти главные герои нашей истории взаимодействуют друг с другом определенным образом.

Когда клиент добавляет товар в корзину, в каталог отправляется запрос на проверку того, есть ли запрашиваемый товар в наличии.

Для отображения общей стоимости корзине нужно знать, какие товары в ней содержатся. Код, который реализует корзину, рассчитывает полную стоимость, отдельно запрашивая у каждого товара его цену и затем высчитывая сумму.

Для окончательного оформления заказа необходимо, чтобы продукты в базе были забронированы. Каталог должен актуализировать наличие товаров, и одновременно с этим корзина должна быть очищена.

Чтобы аннулировать заказ, нужно очистить корзину.

Если вы проанализировали таким образом требования для магазина, то уже разработали простую архитектуру программного обеспечения с модулями и взаимодействиями. Эту архитектуру легко можно отобразить с помощью программных объектов Item (Товар), Cart (Корзина), Customer (Покупатель) и Magazine (Каталог/склад):

```
Article {
    var id;
    var name;
    var price;
}

Cart {
    var articles[];

    function empty() {
        articles = new list();
    }

    function getTotalPrice() {
        var int price = 0;
        for (var i = 0; i < articles.length; i++) {
            price = price + articles.get(i).price;
        }
        return price;
    }

    function addArticle(Article article) {
        ...
    }
}

Customer {
    var cart;
    var store;

    function placeOrder() {
        var articles = cart.articles;
        cart.empty();
        ... // mit den Artikeln dann zum Bezahlen.
        for (var i = 0; i < articles.length; i++) {
            store.removeArticle (articles.[i]);
        }
    }

    function cancelShoppingCart() {
        cart.empty();
    }
}
```

```
Magazine {
    var articles[];
    var numInStore[];

    function removeArticle(Article article) {
        for (var i = 0; i < articles.length; i++) {
            if (articles[i].id == article.id) {
                numInStore[i] = numInStore[i]-1;
            }
        }
    }

    function hasArticle(Article article) {
        ... // Есть ли в базе определенный товар?
    }

    function addArticle(Article article) {
        ... // Добавить в базу определенный товар
    }
}
```

Учитывайте, что каждый созданный вами объект имеет право на существование, потому что он привносит полезные функции, которые нигде до этого не были реализованы. Это право у объектов однозначно есть тогда, когда взаимодействие объекта A с другим объектом изменяет состояние объекта A, состояние другого объекта или состояние обоих — в нашем случае товары могут быть помещены в корзину/удалены из корзины или сняты с учета на складе. Тому, кто читает ваш код, будет понятнее, если корзина сама будет контролировать и изменять свое содержимое, чем если надо будет сначала найти корзину клиента в глобальном двумерном массиве (все клиенты, все товары), а затем изменять в этой корзине статус товаров. При объектно-ориентированном подходе у каждого объекта клиента есть только один объект корзины и объект клиента может выдавать объекты товаров без необходимости разбираться с тем, как именно хранятся объекты товаров. Таким образом, придется иметь дело только с одним простым массивом вместо двумерного.

Если писать приложение, используя объектно-ориентированный подход, то с помощью коротких и просто сформулированных описаний процессов можно в общих чертах набросать взаимодействие объектов, не ломая при этом заранее голову над деталями. Такие грубые наброски могут быть полезны при разработке системы, чтобы вкратце описать фишки и взаимодействие с пользователем. Конечно, все это возможно и в процедурном программировании, но ООП особенно хорошо подходит для сопоставления разговорного описания и реализации, потому что здесь программные объекты могут рассматриваться как ментальные единицы, которые обладают собственной манерой поведения. Например, у нашего объекта Cart (Корзина) есть метод `empty()`, который можно вызвать из `customer.placeOrder()` с помощью `cart.empty()`.

Недостатки и проблемы

Не так хорошо ООП подходит для случаев, когда у вас есть лишь большое количество похожих простых данных, ведь тогда у объектов нет практически ничего, кроме методов, которые отвечают за чтение и запись переменных-членов. Но если вы просто хотите сохранить данные, то это проще сделать с помощью массива или хеширования. А сильно типизированные объектно-ориентированные языки, такие как Java, вынуждают программистов вместо этого определять объекты, даже если введение объекта в данном месте скорее непрактично.

Объекты лучше использовать, когда они:

- демонстрируют различное поведение в зависимости от их внутреннего состояния. Например, объект пользователя, который разрешает определенные действия, только если у него статус «залогинен»;
- могут выполнять вычисления с использованием собственных данных и тем самым создавать сложную систему, которая снаружи проста в применении;
- представляют различные вариации одной темы таким образом, чтобы были основные функции, свойственные всем родственным объектам, и, помимо этого, специфические функции.

Однако при разработке ПО довольно часто приходится иметь дело с однообразными данными, не требующими никаких сложных вычислений или взаимодействий. Такой тип данных хоть и можно представить объектно-ориентированно, но это мало что даст по сравнению с процедурным программированием.

В качестве примера можно использовать оценку годичных оборотов всех филиалов вашей компании за последние годы, то есть временной ряд простых чисел для каждого филиала. Действительно, это можно было бы смоделировать и с помощью объектно-ориентированного подхода: определить объект оценки, объекты филиалов и объекты оборотов, но между этими оборотами нет большого количества взаимодействий. Анализ среднего оборота для каждого из филиалов представляет собой просто считывание количества продаж, их суммирование и выведение среднего значения, при этом статус каждой отдельной продажи не меняется.

Наш придуманный объект оборота никак не развивает систему. Такой объект был бы простым контейнером для чисел, в отличие от объекта товара, который может высчитывать цену брутто из нетто или содержать различные цены для разных конфигураций. Этот тип данных лучше и легче представить в виде массива и функций, например используя процедурное программирование для расчета среднего оборота:

```
var float[][] sales;

function getAverageRevenueForBranch (int branchId) {
    var sum = 0;
    for (var i = 0; i < sales[branchId].length; i++) {
        sum = sum + sales[branchId][i];
    }
    return sum / sales[branchId].length;
}
```

Различные модели объекта в зависимости от языка

Объектно-ориентированный принцип нельзя абсолютно одинаково применить в любом языке. Описанные в данной главе свойства ООП являются более или менее общими, но тем не менее языки существенно различаются в том, как они реализуют эти свойства.

Например, если Java использует для определения объектов своего рода шаблоны — так называемые классы, то другие языки куда более либеральны и в них можно добавлять объектам любые новые свойства во время выполнения программы. Языки, базирующиеся на ООП, в этом плане более жесткие: если захочется добавить объекту новое свойство, будь то метод или переменная, то придется менять определение класса или создавать расширенный класс с помощью наследования.

Обязательно адаптируйтесь к правилам своего языка. Если вы до этого неплохо справлялись с JavaScrip, природа языка которого более динамична, то если вдруг вам понадобится программировать на C++, придется переучиваться. Даже если статический подход таких языков, как C ++, базирующихся на классах, сперва покажется вам сложным и жестким, вы должны или начать использовать их, или в противном случае просто избегать этой языковой семьи. Если же вы, наоборот, хотите перейти с Java на более гибкий ООП-язык, например PHP, то вам следует перестать мысленно делить все на классы и иерархии наследования.

Чем быстрее вы переучитесь и оснастите свои объекты во время выполнения программы новыми методами или переменными экземпляра, тем больше вы будете взаимодействовать с языком. Даже если владеть ООП, при переходе на другой язык возникнет определенный барьер, потому что правила могут быть разными. Снизьте требования к самому себе и откажитесь от идеи во что бы то ни стало перенести свой стиль на новый язык.

Объектно-ориентированное программирование и планы по захвату мира

Объектно-ориентированное программирование похоже на коробку с «Лего»: из маленьких деталей можно собирать большие структуры и соединять их в целые миры. И именно из-за того, что в ООП это выходит проще, чем в процедурном программировании, у многих разработчиков есть период, когда они перебарщивают с объектно-ориентированным моделированием задач. Они пускаются в кошмарные абстрактные иерархии классов даже там, где вероятность того, что этот код понадобится в какой-либо еще части программы, крайне мала.

Новичкам в ООП часто любят давать совет: пишите сначала общий, абстрактный объект, а затем выводите из него конкретные реализации для необходимых ситуаций. Хоть этот совет надежный и подходит для большинства случаев, он может спровоцировать на то, чтобы «затачивать» под повторное использование

даже самые специфические классы, что не имеет смысла за пределами конкретного проекта.

В такой ситуации, будучи не очень опытным программистом, нужно уяснить себе, что задача в том, чтобы написать не перспективную стандартную библиотеку используемого языка, а всего лишь программу, которая, вероятнее всего, не получит широкого распространения. Поэтому поначалу не перебарщивайте с абстрактными классами и наследованием, программируйте только то, в чем есть срочная необходимость. Если вы без особой на то нужды пробуете свои силы в создании объектных иерархий, то зачастую из-за этого не приближаетесь к своей фактической цели, а прокрастинируете, хоть и, казалось бы, эффективным способом. Если у вас есть опыт в подходах к объектно-ориентированному мышлению, то обнаруживать возможности к абстракции у вас будет получаться само собой.

Многие программисты страдают от проблемы «молотка и гвоздя»: если у вас в руке молоток, то все выглядит как гвоздь. Вы используете для любой проблемы один и тот же инструмент, даже там, где это не имеет смысла. После небольшой практики многие незначительные проблемы можно довольно просто решить процедурно, а возможности, которые ООП предлагает для архитектуры, избыточны, если имеешь дело только с парой сотен строк кода. Пока реализуешь такое простое задание с помощью ООП, длина программы может увеличиться в два раза и значительно больше времени уйдет на обдумывание наследования и взаимодействий объектов, чем на саму проблему.

Поэтому, если вы пишете на языках, которые не требуют объектной ориентированности, таких как PHP или JavaScript, то вы пишете в соответствии с проблемой: небольшие программы — процедурно, большие проекты — с самого начала объектно-ориентированно. Если же вы используете такой язык, как Java, то этот совет мало чем поможет, потому что процедурно на Java писать затруднительно, да это и не предполагается. В таком случае сохраняйте спокойствие и не тратьте много времени на мудреную архитектуру.

24 Хранение данных

> Если вам срочно требуется таблица, в которой не более семи столбцов, которую удобно обрабатывать вместе с другими людьми, то вы, кстати, можете просто-напросто воспользоваться календарем Google.
>
> *Комментатор в «Блоге для садоводов», riesenmaschine.de*

У некоторых программ нет высоких требований к хранению данных. Если вы, к примеру, пишете чат, то должны позаботиться лишь о файлах конфигурации и в случае необходимости — о журналировании сообщений. В общем и целом основной задачей программы чата является только получение сообщений, их отображение и возможность рассылки собственных сообщений.

Для многих игр не требуется сложного управления данными. Графика и скрипты прилагаются, а текущее состояние прохождения игры можно легко сохранить в небольших файлах. Разработчикам игр редко приходится беспокоиться об обмене данными с другими программами или о подходящих форматах для долговременного архивирования.

Разработчикам текстовых редакторов в этом отношении тоже повезло: пользователи открывают файлы, редактируют их и сохраняют, программа-редактор не должна предоставлять базу данных. Пользователь сам полностью отвечает за то, где и как он хочет хранить свои данные.

Но во многих сферах эта ситуация выглядит иначе, и вам, как разработчику, приходится думать о том, как управлять данными, делать их доступными для поиска и, возможно, архивации. Некоторые приложения, такие как ПО для научного анализа, совершенно бесполезны без своих данных. Здесь содержимое зачастую важнее, чем сама программа, и вы, как разработчик, довольно часто являетесь единственным или одним из немногих пользователей, так что вам приходится сразу, в процессе разработки обдумывать, помимо прочего, еще и вопрос хранения данных.

Если вы пишете мобильное приложение, то обычно у вас есть back-end-база данных, которая берет на себя управление пользователями и которая — в зависимости от типа приложения — должна уметь сохранять и отправлять геоданные или введенную пользователем информацию. Именно приложения для смартфонов дают сегодня возможность почувствовать на собственной шкуре проблемы прошлых поколений программистов: устройства относительно медленные, и у них мало памяти. Поэтому, как разработчик, вы не можете просто держать все данные

в RAM, а должны вместо этого выгружать ненужные данные во флеш-память. Так как подключение к back-end-базе через UMTS гораздо более медленное, чем через компьютер с DSL- или WLAN-подключением, то очень может быть, что вам придется усиленно подумать над тем, какими данными вы будете обмениваться между мобильным приложением и back-end, а какие данные останутся на устройстве. В то же время самая важная информация должна храниться на back-end-сервере, потому что смартфоны довольно часто теряются. И ваши пользователи будут вам благодарны, если после этого все данные не пропадут безвозвратно, а их можно будет просто синхронизировать на новом устройстве.

Вам следует заранее принять решение о том, как будут храниться ваши данные, потому что потом поменять его будет трудно. В принципе, это возможно — прорефакторить приложение таким образом, чтобы оно сохраняло свои данные в базе, а не в файлах, но это изменение может в дальнейшем привести к значительным затратам. Поэтому, перед тем как начать программировать, вам стоит обдумать свои требования к хранению данных.

> Движок блога, который я написала сама и с помощью которого веду несколько блогов, не использует базу данных, потому что мне до сегодняшнего дня по какой-то неясной причине неприятны базы данных. Хотя я их уже долго использую и в данной ситуации тоже придумала бы, как теоретически это могло бы работать. О принятом много лет назад легкомысленном решении «ах, это точно пойдет и с текстовыми файлами» я часто сожалею до сих пор, потому что, конечно, базы данных — это решение, которое было разработано как раз для таких задач.
>
> *Катрин*

Для принятия этого решения не столько важно, как вы храните данные (что очень легко), сколько каким образом вы хотите повторно их запрашивать. Данные, которые требуется просто заархивировать, можно сохранять и в файлах. Но если вы хотите просматривать данные, вычислять сумму, фильтровать и объединять массивы данных, то вам стоит подумать о базе данных.

Существуют три различных способа хранения данных:

- файлы. Это простейшее решение для небольших проектов и их отладки;
- базы данных. Вы не пройдете мимо баз данных, если ваша программа должна осуществлять поиск или анализ данных;
- системы управления версиями. Довольно необычный способ хранения данных, но он может быть полезен, если содержимое часто меняется.

Файлы

Ваша программа без особых усилий может сохранять свои данные в файлах и позже прочитывать их, правда, только если прямо сейчас вы не пишете чистое веб-приложение с JavaScript.

Часто вы будете получать данные уже в файлах, будь то CSV-файлы с финансовыми данными, которые вы должны импортировать и проанализировать, или

результаты научных измерений, которые вы хотите обработать. Но даже если вы берете данные из Интернета или ваши пользователи сами создают данные, вы можете помещать их в файлы и позже снова прочитывать.

У хранения данных в файлах есть несколько преимуществ. Во-первых, этот способ не зависит от языка программирования. Во-вторых, вам не приходится разбираться с соединениями баз данных, SQL или системами контроля версий. Архивация данных тоже не является проблемой, ее, помимо прочего, может обеспечивать программа резервного копирования данных, если такую предоставляет ваша система. Обмениваться данными, даже очень большими, можно по электронной почте, через Dropbox или SFTP-загрузку. На крайний случай у нас все еще остаются USB-флешки.

Но у этого способа есть и несколько недостатков: полнотекстовый поиск оперативен в использовании и быстро выдает результаты, но только до тех пор, пока у вас лишь небольшие объемы данных. А вот для больших файлов потребуется установить поисковую машину вроде Lucene и работать с ней из вашей программы.

Если вы захотите поддерживать гибкие запросы, вроде «все магазины с оборотом между x и y евро», то должны будете считать всю информацию и написать код для каждого запроса такого типа. А создание в случае необходимости множества сложных и структурированных поисковых запросов будет трудоемким («все измеренные значения, которые относятся к устройству Robot 183, и все измеренные значения Joint 3 rotation, которые больше 45°»). С базой данных это проще, потому что базы данных и SQL были изобретены как раз для этой цели.

А что, если у ваших пользователей есть возможность менять данные? Тогда вам необходимо загружать пакеты данных и снова вносить измененные данные в файл. И тогда надо думать о том, каким образом вы хотите распорядиться этими изменениями. Каждый раз переписывать всю базу данных? Управлять версиями? Это тоже может быть довольно неудобно и будет проще осуществить в базе данных или в системе контроля версий. Тут же вас подстерегает и самая труднорешаемая проблема хранения в файлах: если два пользователя одновременно вносят изменения, которые затрагивают различные части одного и того же файла, ваша система должна свести эти изменения без потери результатов работы каждого из пользователей. Это может считаться каким угодно, но только не простым, и в итоге специально для решения этой проблемы была изобретена система управления версиями. В базе данных блоки данных хранятся в различных таблицах (реляционные базы данных) или записях (документоориентированные базы данных), и база отвечает за то, чтобы при параллельных изменениях не возникали конфликты.

Если вы решили хранить данные в файлах, то вам необходимо выбрать формат. В настоящее время широко используются CSV/TSV, XML, JSON или свободные текстовые форматы. Какой бы формат вы ни выбрали, заранее убедитесь в том, что данные закодированы в UTF-8, иначе при прочтении вас будут ждать неприятные сюрпризы (см. главу 17).

А сейчас для каждого из форматов мы приведем пример из системы управления ресурсами (ERP), в котором в одном файле должны содержаться свойства двух предметов: дивана и стола.

Свободные текстовые форматы предлагают абсолютную свободу действий: вы сами решаете, как хранить данные. Например, можете размещать по одному блоку данных в каждой строке.

Такое хранение подходит для журналов и других похожих форматов, потому что записи в журнале всегда затем прикрепляются к файлу журнала.

Это происходит так быстро, что программа продолжает работу, несмотря на прикрепление к файлу журнала. Кроме того, мы рекомендуем отказаться от использования собственных форматов, потому что вам придется самостоятельно писать весь код и вы не сможете воспользоваться библиотеками, которые доступны, например, для XML.

Пример, в котором различные товары разделяются строкой `@@item@@`:

```
@@item@@
name: couch
color: slab grey
dimensions: 200x90x70
@@item@@
name: table
color: wood
dimensions: 150x100x90
```

CSV/TSV

CSV/TSV — это текстовые форматы, которые могут быть импортированы в программы электронных таблиц, такие как Excel, и экспортированы из них. Строки разделены разрывом, а столбцы — запятой или знаком табуляции (CSV или TSV). CSV/TSV-файлы довольно компактны. Этот формат является хорошим выбором прежде всего тогда, когда вы хотите просматривать данные. В этом случае вы сможете избавиться от необходимости писать программу для просмотра данных, а вместо этого получите возможность сразу открыть их в программе электронных таблиц.

В нашем примере первая строка содержит названия столбцов, а две следующие строки — товары:

```
item;color;length;width;height
couch;slab grey;200;90;70
table;wood;150;100;90
```

Проблемы начинаются тогда, когда запись содержит разделители строк или столбцов (запятую, точку с запятой или табуляцию). Тогда вам нужно или отфильтровать эту запись, или как-то перекодировать ее, потому что иначе функции чтения данных перепутаются. Иногда для удержания содержимого в столбцах, в которых есть разделитель, могут использоваться кавычки, но тогда они должны быть перекодированы.

CSV/TSV непригодны для хранения иерархических данных, потому что эти форматы предусматривают жесткий шаблон из строк и столбцов, совсем как в листе электронной таблицы. А для иерархических данных должна быть возможность создавать дальнейшее подразделение в этой же ячейке, но здесь это невозможно.

XML

XML — это замечательный формат для хранения сложных и иерархических данных. В XML вы можете с легкостью сохранить не только имя клиента и адрес, но и все заказы. Многие текстовые редакторы используют XML также для хранения текстов с форматированием. Как для чтения, так и для записи XML существуют библиотеки практически для всех языков, что облегчает вам жизнь. Обязательно используйте какую-нибудь из этих библиотек, потому что тогда вам не придется беспокоиться о том, как сохранить < или & — оба этих знака входят в структуру XML и, следовательно, не могут просто так появиться в пользовательских данных. Библиотеки XML заботятся о том, чтобы замаскировать эти знаки таким образом, чтобы получился рабочий XML-файл, но при этом сами знаки не потерялись.

Наша мебель в XML:

```
<item name="couch">
    <color="slab grey">
    <dimensions>
        <length>200</length>
        <width>90</width>
        <height>70</height>
    </dimensions>
</item>
<item name="table">
    <color="slab grey">
    <dimensions>
        <length>150</length>
        <width>100</width>
        <height>90</height>
    </dimensions>
</item>
```

Из примера можно ясно увидеть, что `<dimensions>` является контейнером, в котором информация о длине, ширине и высоте сведена к одной единице. Это удобно для хранения больших объемов информации в программе в хеше.

Тем не менее существует и следующее правило: XML часто бывает лишним, если вы хотите хранить только простые данные. В то же время он не предлагает функций поиска и анализа баз для очень сложных данных. Если на первый взгляд использование XML кажется вам целесообразным, то по крайней мере задумайтесь ненадолго над тем, не была бы база данных лучшим решением, чем это.

JSON

Формат JSON особенно популярен в Web 2.0, который берет свое начало в JavaScript. Он предлагает те же возможности, что и XML, — хранение иерархических и сложных данных, но при этом он более гибкий и данные из него считываются быстрее, по крайней мере в JavaScript. JSON часто используется для обмена данными между разными компьютерами, а вот как формат хранения он в настоящее время используется реже. Наш пример в JSON выглядит следующим образом:

```
[
{
    "type" : "item",
    "name" : "couch",
    "color": "slab grey",
    "dimensions": {
        "length": 200,
        "width": 90,
        "height": 70
    }
},
{
    "type" : "item",
    "name" : "table",
    "color" : "wood",
    "dimensions": {
        "length": 150,
        "width": 100,
        "height": 90
    }
}
]
```

Как и XML, JSON умеет хорошо описывать сложные структуры: в примере длина, ширина и высота сохранены как частный объект `Dimension`, а не как отдельные свойства.

Как и при использовании XML, если вы захотели хранить свои данные в JSON, то подумайте о том, не было бы целесообразнее использовать базу данных. Для хранения JSON особенно хорошо подходят документоориентированные системы управления базами данных (СУБД), о которых мы расскажем далее, потому что данные в них хранятся именно в формате JSON. Но и из реляционных баз данных можно легко генерировать JSON и использовать для веб-приложений.

YAML

YAML (от YAML Ain't Markup Language — «YAML — не язык разметки») — это формат хранения данных, который представляет собой расширенный JSON. YAML может включать комментарии, упорядоченные списки и пользовательские типы данных. Приятная новость: строки не нужно заключать в кавычки, поэтому текст читается легче. Любой файл в формате JSON является корректным файлом в формате YAML, но не наоборот. Наш пример в YAML:

```
---
type: item
name: couch
color: slab grey
dimensions: {length: 200, width: 90, height: 70}
---
type: item
```

```
name: table
color: wood
dimensions: {length: 150, width: 100, height: 90}
```

Несмотря на то что многие библиотеки обеспечивают YAML хорошую поддержку, для многих языков это относительно редкий формат.

Системы управления версиями

В системах управления версиями тоже хранятся файлы, поэтому, как и при хранении в системе данных, приходится думать о выборе формата. Большим преимуществом использования системы контроля версий является то, что хранение данных отделено от долговременного архивирования, а для архивации применяется проверенная система, которую не приходится разрабатывать самостоятельно.

Системы управления версиями довольно редко используются как back-end-базы хранения данных. Они не помогают разработчику ни осуществлять широкий поиск, ни делать расчеты в данных. Но они могут быть полезны в отдельных случаях, а именно, когда пакеты данных часто меняются и нужно сохранять старые версии. Это можно делать и в базе данных, но неудобно. Возможными сферами применения систем управления версиями могут быть движки «Википедии» или блогов.

Базы данных

За все эти годы появилось довольно много различных типов баз данных. Но на протяжении многих десятилетий реляционные базы данных (пояснение термина см. далее) были единственно значимыми, и сегодня мы подразумеваем этот тип, даже если говорим просто о базах данных.

В 2000-х наряду с хорошо зарекомендовавшим себя реляционным стандартом развивались и другие, которые были объединены ключевым словом NoSQL. В некоторых сферах применения они являются интересной альтернативой реляционным базам данных. Что их отличает от реляционной модели, мы сейчас объясним.

Искать и найти

Базы данных круты не потому, что вы можете размещать в них данные, ведь так же хорошо это работает и с файлами. Базы данных куда более блистательны в вопросах гибкого поиска в массивах данных.

Например, вы хотите сохранить статьи из системы управления ресурсами в виде XML-файлов, тогда вы сможете сортировать отдельные файлы по времени их появления, для чего нужно будет создать иерархию каталогов «дата/месяц/день» и использовать временную метку в качестве имени файла. Или вы могли бы создать для каждого автора отдельную папку с его статьями, или смешанную форму с датой/автором/названием статьи. Во всех случаях потребовалось бы довольно много времени для того, чтобы найти критерий, который не закодирован в иерархии

каталогов или в имени файла, потому что вам пришлось бы прочитать все файлы и искать в том поле XML, которое соответствует вашим критериям поиска.

Базы данных, напротив, сохраняют данные структурированно и позволяют осуществлять поиск по всем свойствам массива данных. У вас есть свобода выбора: искать по автору, по дате или по такой комбинации, как «статьи, которые были написаны Йоханнесом в период с 29.01.2009 по 13.10.2013». Базы данных могут быстро выполнять такой гибкий поиск, и процесс пойдет еще быстрее, если вы перед этим создадите индексы в массиве данных.

В основе своей индексы работают по принципу оглавления книги. Когда вы ищете в научно-популярной книге страницы, на которых используется определенный термин, то не читаете книгу от начала до конца и не отмечаете найденные страницы, а вместо этого смотрите в словарь терминов в конце книги.

И поэтому, если вы используете базы данных, то прежде стоит немного узнать об индексах, а затем определить индекс для каждого свойства, которое вы, вероятно, захотите найти. В реляционных базах данных, о которых мы расскажем в следующем разделе, индексы для первичных ключей создаются автоматически. Для всех остальных свойств вы должны создавать в базе данных дополнительный индекс. В системах NoSQL вы должны создавать практически все индексы явно. База данных постоянно и автоматически актуализирует индексы: если вы добавляете новые данные, то они сразу индексируются. Если изменяете данные, то база данных тут же обновит соответствующий индекс. Если вы заметили, что необходимы дополнительные индексы, то можете добавить их позднее. Но не создавайте индексы для всех свойств, а ограничьтесь теми, которые нужны вам прямо сейчас, и добавляйте последующие индексы в том случае, если заметите, что некоторые запросы обрабатываются очень медленно.

Реляционные базы данных

В реляционных базах данных (РСУБД) в качестве основной системы регулирования выступают таблицы. Все данные содержатся в таблицах, которые упорядочены как раз так, как мы это себе интуитивно и представляем: в графы (Columns) и строки (Rows). Между записями в различных таблицах могут устанавливаться связи (Relation), это всегда происходит построчно и приводит к построчным результатам.

Реляционные базы — это ваш выбор, если у ваших данных всегда один и тот же формат, у всех записей одни и те же свойства (ну примерно как имя и фамилия у человека). Реляционные базы хороши в управлении объемами данных от больших до огромных, они могут по записям распознавать определенные пользователем критерии, очень быстро фильтровать и выдавать их. Хуже РСУБД подходят для хранения данных с меняющимися свойствами, как в электронных адресных книгах, где, помимо фиксированных свойств, пользователь может добавлять к каждой записи собственные заметки. Кроме того, РСУБД не очень подходят для хранения сложных структур данных, таких как HTML-страницы с их нагромождениями тегов.

Какие свойства должны отображаться в таблице, вы определяете сами, когда создаете так называемую схему базы данных. Эта схема — не что иное, как определение того, как данные хранятся в базе данных:

```
CREATE TABLE USER (
    id INTEGER NOT NULL,
    firstname VARCHAR(30) NOT NULL,
    lastname VARCHAR(30) NOT NULL,
    department INTEGER,
    PRIMARY KEY (`id`)
);
```

Посредством этой команды вы создаете таблицу `User` с четырьмя столбцами, при этом каждая запись должна иметь идентификатор, имя и фамилию, а также отдел.

Вы можете написать определения этих схем вручную или воспользоваться одним из многочисленных графических front-end, который совместим с определениями таблиц, индексами и частично запросами в базе данных. Без сомнения, наиболее известным front-end является Access Microsoft, но он привязан к базам данных Microsoft. Для других реляционных баз данных существуют аналогичные инструменты.

Часто таблицы создаются таким образом, что идентификатор каждой записи содержится в первом столбце (свойство), что делает этот идентификатор первичным ключом (ПК) таблицы. У первичных ключей есть особенности, среди которых следующие: они не могут получить значение `NULL`; в таблице не может быть двух строк, которые содержат один и тот же ПК. Таким образом, ПК — это уникальный ID (см. раздел «ID, GUID, UUID» главы 26).

Реляционные базы данных популярны из-за того, что в них вы, будучи разработчиком, можете распознавать повторы данных и выгружать информацию в собственные таблицы. В приведенном ранее примере `department` (то есть отдел, в котором работает сотрудник) представляет собой всего лишь число, хотя у самого отдела есть имя, он занимает одно или несколько зданий, комнат и т. д. Но если вы не хотите снова и снова вносить эту информацию в таблицу `User` для каждого сотрудника, то вместо этого нужно создать таблицы `Department`, `Address` и т. д. Тогда число в таблице `User` соответствует первичному ключу таблицы `Department` и определяется как *внешний ключ*. Эта система, позволяющая создавать связи (Relation) через таблицы посредством связей внешних и закрытых ключей, и дала реляционным базам данных их название[1].

Язык реляционных баз данных — это SQL. SQL может заполнять базу данных информацией, считывать, изменять и удалять данные. Помимо этого, SQL предлагает еще целый ряд операторов, таких как суммирование, агрегирование, сортировка, но наиболее важными из них являются следующие:

- `INSERT` — позволяет построчно добавлять данные в таблицу;
- `SELECT` — позволяет возвращать выборку из базы данных;
- `UPDATE` — с помощью этого инструмента можно изменять содержимое отдельных ячеек в таблице;

---

[1] Как удалить дубликаты путем нормализации своей схемы базы данных, мы описывали в главе 15.

- `DELETE` — построчно удаляет данные.

Хоть ячейка и является наименьшей единицей в реляционной базе данных, то есть значением на пересечении столбца и строки, тем не менее большинство операторов созданы для строк. И поэтому, если писать:

```
SELECT firstname, lastname FROM USER;
```

то получаешь обратно имена и фамилии из всех строк базы данных.

Существуют также операторы для столбцов, например:

```
SELECT DISTINCT lastname FROM USER;
```

С помощью этого оператора вы получаете список всех фамилий, причем дубликаты уже удалены.

SQL является декларативным языком (см. раздел «Языковые семьи» главы 26), что означает, что вам не требуется (хотя вы и можете) писать циклы.

Когда вы пишете на SQL, то указываете, какой оператор (то есть `SELECT` или `INSERT`) должен применяться к какой таблице, и еще можете дополнительно указывать критерии фильтрации:

```
SELECT * FROM USER WHERE lastname = 'Meier'
```

В императивном языке вам пришлось бы писать цикл, который перебирал бы массив `User`, исследовал каждую запись и, если бы она содержала свойство «фамилия» со значением «Майер», вставлял эту запись во второй массив. Этот второй массив вы бы потом вернули:

```
function findUsersByLastName(usersIn, lastName) {
    var resultUsers = [];
    for (var i = 0; i < usersIn.length; i++) {
        if (usersIn[i].lastName == lastName) {
            resultUsers.push(usersIn[i]);
        }
    }
    return resultUsers;
}
```

Реляционные базы данных традиционно хранятся на сервере, а для разработок используется SQL-клиент. Это программа, которую вы устанавливаете на локальном компьютере и с помощью которой подключаетесь к базе данных. В этой программе вы можете вводить SQL-операторы и напрямую просматривать результаты, выданные базой данных. Если хотите запрашивать РСУБД непосредственно из своей программы или хранить информацию в базе данных, то используйте библиотеку, которая будет обращаться к этой базе данных.

Кроме модели «клиент/сервер», которая родом еще из времен больших датацентров, хорошо зарекомендовала себя как локальная база данных SQLite. SQLite играет особо важную роль в разработке мобильных приложений, где отправлять каждый SQL-запрос на сервер было бы слишком медленно. Плюс в том, что если вы знаете SQL, то можете хранить свои данные совсем как на сервере баз данных. Но поскольку это происходит локально на смартфоне, то все работает гораздо быстрее. SQLite отлично подходит также для тех, кто смело хочет экспериментир-

ровать с хранением данных из приложения в базе данных. Сервер вам не нужен — все происходит быстро, а если вы по невнимательности перепутали данные в базе данных, то можете легко восстановить предыдущее состояние, потому что SQLite хранит все данные в локальных файлах, которые можно просто копировать и архивировать.

Базы данных NoSQL

Реляционные базы были хорошим решением ровно до тех пор, пока требовалось обрабатывать одни и те же данные с фиксированной структурой. Если бы больничная касса захотела управлять своими клиентами: их распределением по филиалам, их медицинскими счетами и платежами, то реляционная база была подходящим выбором, потому что она позволяет записывать определенные одинаковые свойства, такие как имена и адреса, для каждого клиента кассы под номером участника. Все врачи и филиалы получают идентификаторы, и тогда можно установить связи между клиентом и доктором: счет 49 378 на 129,87 евро для участника 72 627 выдан врачом 2625. Статистика удельных платежей клиентов, расходов на врачей и т. п. работает относительно быстро и проста в разработке.

Такой тип приложений был и остается важным. Он будет существовать и в дальнейшем и, вероятно, станет реализовываться на основе реляционных баз данных. Но с появлением Интернета на безоблачном горизонте господства реляционных баз стали собираться тучи. Системы управления содержимым и «вики» работают не со многими структурно одинаковыми пакетами данных, а с относительно небольшим количеством сильно структурированных внутри данных. Жесткий и схематичный способ хранения в реляционных базах плохо подходит для документов, в которых могут встречаться одна или несколько таблиц, изображения, ссылки и различные заголовки. А если у пользователя к тому же есть право придумывать собственные элементы, например вставленные в текст объекты с определенным форматированием, то работа разработчика становится ужасающе трудным делом.

У возникавших тогда новых поисковых систем и социальных сетей, напротив, были несколько противоположные нужды, которые тоже не слишком хорошо решались с помощью реляционной модели: нужно было уметь управлять огромными объемами данных и отвечать на запросы за доли секунды. Для таких приложений страница запросов реляционной базы была неоптимальной, поскольку эти базы данных были разработаны, чтобы отвечать на запросы вида «выдай мне актуальные обороты всех филиалов за последние 12 месяцев». Хоть реляционные базы и способны поддерживать полнотекстовый поиск, который требуется для поисковых систем, но их производительность неидеальна для этого. И хотя связи между пользователями социальных сетей также могут быть сохранены в реляционных базах данных и в них можно выполнять такие запросы, как «выдай мне всех пользователей с интересами, как у пользователя Яндера», все же SQL не очень хорошо подходит для программирования таких запросов.

Поэтому в конце 1990-х и начале 2000-х были разработаны новые концепции баз данных, которые больше не полагались на реляционную модель и не использо-

вали SQL для запросов, их объединили под кратким названием NoSQL. Поскольку эти базы данных были созданы в то время, когда Интернет уже стал обычным делом, то в них для запросов и хранения данных ставка была сделана на веб-технологии. Таким образом, отпала необходимость устанавливать для разработки клиент для БД, как это нужно было делать с РСУБД, и можно работать напрямую с веб-браузером.

Если потом нужно будет обратиться к этим базам данных, можно вызвать библиотеку вроде cURL из собственных функций или вызвать напрямую библиотеку, разработанную под используемую NoSQL-базу данных.

Существует целый ряд баз данных NoSQL, но из них для разработчиков-одиночек и небольших групп особенно интересны следующие две.

Документоориентированные базы данных, такие как CouchDB или MongoDB

Они позволяют сохранять данные с очень гибкой структурой. Это идеально подходит для сайтов или систем управления контентом, потому что не приходится мучиться над тем, как отобразить структуру документа в базе данных.

Пользователь может по своему желанию без особо трудоемкого программирования добавлять к документу абзацы, или ссылки, или абсолютно новые элементы, которые изначально не предусматривались.

У таких баз данных нет ни таблиц, ни столбцов, ни SQL. Вместо этого в них хранятся JSON-документы, которые могут содержать дочерние элементы. И дочерние элементы, в свою очередь, могут содержать другие дочерние элементы. Как разработчик, вы пишете небольшие функции JavaScript для выполнения запросов.

А если вы хотите дать посетителям возможность комментировать посты в блоге? В реляционной модели вам для этого пришлось бы создать таблицу `COMMENT`, содержащую комментарии посетителей; затем еще одну таблицу, `VISITOR`, в которой зарегистрирована информация о каждом посетителе, например имя и электронный адрес, и, наконец, связующую таблицу между `COMMENT` и `VISITOR`, чтобы отследить, какой посетитель какой комментарий оставил. И между этими тремя таблицами и таблицей для постов вам пришлось бы создать еще одну связь. В документоориентированной модели это гораздо проще: комментарии могут быть просто прикреплены к каждому посту в виде дополнительных дочерних элементов, и при этом у вас остается возможность путем сравнительно небольших усилий заблокировать сообщения спамеров.

Такой тип хранения гораздо ближе к нашему образу мышления: комментарии к одному посту «принадлежат» этому посту и подчиняются ему. Однако реляционная модель требует, чтобы комментарии хранились отдельно от поста и затем вновь объединялись с ним через связи.

Документоориентированные базы — это хороший вариант в том случае, если ваши данные в структуре динамичны или если вы к началу проекта еще не знаете точно, каким образом данные будут структурированы позднее. Если же изменяется только содержимое, а структура практически та же, тогда у документоориенти-

рованных баз нет больших преимуществ по сравнению с реляционными базами данных.

Даже возможность агрегированной оценки (сумма продаж всех филиалов, средняя заработная плата сотрудников с максимальным и минимальным значениями) не является сильной стороной этих баз, потому что для этого были созданы РСУБД.

Графовые базы данных, такие как Neo4j

Если речь идет главным образом о создании связей между пакетами данных — неважно, будь то пути химических реакций, денежные потоки или социальные сети, — то хорошим выбором будут графовые базы данных. Они могут использоваться, например, в рекомендательных системах, как у Amazon: если система моделирует заказчиков и товары в виде узлов, а покупки — как связи между узлом клиента и узлом товара, тогда становится довольно просто на основе покупок клиента А найти других клиентов с похожими покупательскими привычками. Когда система проверяет их покупки, она может сравнивать их с покупками клиента А и предлагать этому клиенту те товары, которые он сам до этого не покупал, но которые охотно покупали клиенты с похожими потребностями.

У графовых баз данных исключительно спартанская модель: есть только узлы и связи. Узлы могут содержать такие свойства, как ID, имя или определенные разработчиком и свободно выбранные им свойства. Соединения, как правило, подкрепляются глаголами вроде knows, likes (в социальных сетях) или inhibits, catalyzes (в науке).

Созданное Facebook приложение Graph search довольно просто реализуется посредством графовой базы данных. С его помощью можно просматривать свои контакты и делать запросы вида «найти друзей моих друзей, которые часто посещают группу “Вязаный единорог” и любят музыку Джонни Кэша».

Кроме того, SQL не производит поиск в графовых базах данных. Частично у них есть SQL-подобный язык запросов, но отчасти вы должны программировать запросы сами.

Неважно, на какой тип хранения данных вы решитесь, — принимайте это решение осознанно, а не на основе неясной неприязни к конкретной технологии («XML — ну это просто про-ошлый век!») или страха, что не владеете альтернативными технологиями.

Хоть изучение новых технологий баз данных и потребует времени и усилий, но в долгосрочной перспективе гораздо более трудоемким может оказаться многолетнее мучение с совершенно не подходящей системой.

Каждый следующий фрагмент кода, который вы пишете, чтобы удержать свою систему на плаву, затрудняет переход к лучшему варианту.

25 Безопасность

Я не сажаю земляные орехи, когда за мной наблюдает обезьяна.

Пословица народа тив

Эта глава предназначена в первую очередь для тех читателей, которые собираются писать веб-приложения. Лазейки в системе безопасности есть повсюду, но пользуются ими прежде всего тогда, когда выполняется одно из трех условий: речь идет о больших деньгах, есть возможность использовать чужие данные для хулиганских выходок или можно воспользоваться чужим сервером для нехороших вещей, таких как рассылка спама. Используемый каким-нибудь трудовым коллективом инструмент для анализа данных или частная программа для управления домашним хозяйством вряд ли возбудят в ком-то желание воспользоваться уязвимостями, которые, возможно, в них имеются. Быть может, взломом офисных приложений еще занимаются в качестве учебного задания в начальных школах для хакеров, но те, кто уже вырос из пеленок, не интересуются этим всерьез.

Да, известия о том, что в системе безопасности офисных приложений, таких как Microsoft Office или Adobe Reader, выявлены бреши, снова и снова попадают в заголовки новостей. Но здесь речь идет о ПО, которым пользуются миллионы людей, что делает его привлекательной целью для атак. Что касается вашей самопальной программы для научных отчетов, самостоятельно написанного судоку или маленького вспомогательного скрипта, то весьма вероятно, что количество их пользователей никогда не превысит пару десятков. А ПО для торговли акциями или интернет-банкинга вы, как надеются авторы, пока что разрабатывать не собираетесь.

Дела обстоят иначе, если вы реализуете небольшой веб-проект. Подавляющая часть уязвимостей, которыми пользуются злоумышленники, сосредоточена в ПО, имеющем дело с Интернетом (например, в браузерах и их плагинах), или в веб-приложениях, которые предоставляют услуги в Интернете (на веб-форумах и в системах управления контентом). Даже если число пользователей вашего веб-приложения остается маленьким, его может применить практически любой интернет-пользователь. При этом, возможно, само приложение хакерам глубоко неинтересно, но оно — входные ворота, через которые другие люди получают доступ к вашему серверу. Затем злоумышленники воспользуются им для рассылки спама или сделают его командным центром в Третьей мировой войне.

Не убеждайте себя, что ваше веб-приложение настолько незначительное и неизвестное, что ничего плохого случиться не может. Есть множество технически одаренных людей, отличающихся непреодолимой тягой к наживе либо скверных по натуре, и рано или поздно кто-нибудь из них возьмется за ваше приложение и проверит его на наличие уязвимостей.

Даже внутренние приложения фирм, которые доступны только в LAN, следует внимательно проверить на случай худшего. Нет ли у сотрудников неконтролируемого доступа к любым массивам данных? Что произойдет, если какой-нибудь особенно мотивированный сотрудник взломает приложение? Тот факт, что в результате он навсегда будет отстранен от должности, может утешить только в том случае, если он не прихватил с собой список ваших клиентов или другие внутренние данные. Такие происшествия обычно не получают огласки. Однако некоторые сенсационные случаи попадают в СМИ, например нашумевшее воровство данных в лихтенштейнском банке LGT Treuhand в 2008 году. Преступником оказался внештатный сотрудник, которому была поручена проверка отсканированных документов. А аналитик секретной службы Эдвард Сноуден, который в 2013 году сделал достоянием общественности значительную часть информации об интернет-слежке, которую вело Агентство национальной безопасности США, смог сделать это только потому, что у него был доступ к многочисленным PowerPoint-презентациям для внутреннего пользования, рассказывающим о программах слежки.

Собственно говоря, понятие «безопасность» вводит в заблуждение. Там, где происходит взаимодействие с внешним миром, не бывает стопроцентной безопасности. Человеческое тело уже несколько миллионов лет пытается справиться со следующей задачей: с одной стороны, получать из окружающей среды воздух и питательные вещества, с другой — не пропускать возбудителей болезней. И несмотря на это, люди время от времени погибают от инфекций. Неважно, будете ли вы пользоваться только элементарными стандартами, о которых пойдет речь в этой главе, или потратите пару недель на более основательное изучение этой области, результатом будет ни в коем случае не безопасность, а лишь снижение рисков. Из того факта, что даже в системах безопасности крупных компаний обнаруживается большое количество дыр, не следует делать вывод, что в IT-отделах там сидят одни дурни. Просто дыр всегда будет больше, чем заплат.

Итак, речь не идет об обеспечении безопасности — это фикция. Ваша цель должна заключаться в том, чтобы не быть слишком уж уязвимой мишенью для атаки. На свете нет такого велосипедного замка, который нельзя было бы взломать, однако хороший замок может вынудить потенциальных воров угнать вместо вашего велосипеда чей-нибудь другой.

Важные концепции

Низкий уровень безопасности связан не только с незнанием или нехваткой времени. Даже не самые опытные программисты знают, что, вообще говоря, было бы очень неплохо уделять вопросам безопасности чуть-чуть больше внимания. Вероятно, им также известно, в каких местах их код дает повод тревожиться. Но никто не любит

заблаговременно готовиться к беде. Хотя мысленное соприкосновение с возможными неприятностями и менее болезненно, чем появление реальных проблем, однако очевидная альтернатива — ничегонеделание — все же более комфортна.

Эта проблема тянется через всю жизнь и простирается от пренебрежения регулярным резервным копированием и контрольными посещениями зубного врача до нежелания делать пенсионные отчисления. В исследовании 1974 года, проведенном в связи с введением в Германии обязательного использования ремней безопасности, говорится: «Ремень безопасности в первую очередь ассоциируется с опасностью аварии и ее последствиями и лишь во вторую — с его непосредственной технической функцией — защищать от этих опасностей». Поэтому испытуемые, слыша слово «пристегнуться», «с психологической точки зрения попадают в тиски. С одной стороны, они понимают, что с ремнем ездить безопаснее, с другой — ремень безопасности актуализирует в них страх, которого они хотели бы избежать. Попытка избежать страха не позволяет им эффективно избегать опасности»[1].

На тот случай, если вы не дочитаете эту главу до конца, так как не любите слишком много думать о брешах величиной с футбольные ворота в безопасности вашего кода, мы сразу перейдем к центральному пункту этой главы: безопасность — это тема, известная своей каверзностью, в которой даже самые лучшие вновь и вновь совершают фатальные ошибки. И будучи новичком, и даже став продвинутым программистом, вы все равно наверняка будете постоянно совершать некоторые ошибки. Для всех аспектов разработки ПО, важных с точки зрения обеспечения безопасности, существуют готовые решения. Применяйте их (см. также главу 19).

Если вы решили писать надежный код, вам придется следовать правилу: недостаточно потратить время один раз, вы должны постоянно вкладывать свое время, чтобы оставаться на нынешнем уровне. То, что было сегодня решением средней надежности, завтра может уже не быть таковым. Ознакомьтесь с материалом, который Open Web Application Security Project («Открытый проект обеспечения безопасности веб-приложений») собрал для вас на своей странице www.owasp.org. Особое внимание обратите на «шпаргалки» — Cheat Sheets (www.owasp.org/index.php/Cheat_Sheets), чтобы получить представление о масштабах задачи, за решение которой вы беретесь.

И даже если вы отказываетесь от того, чтобы писать все самому, это не значит, что вам вообще ничего не нужно знать о безопасности. Вы должны знать, где скрываются распространенные проблемы с безопасностью — в каких местах требуется применять правило «не делай сам». Имеет смысл понимать используемый вами готовый код и его readme-файлы хотя бы в общих чертах. А если в какой-то ситуации применить готовые решения действительно будет невозможно, то после прочтения этой главы вы хотя бы будете в состоянии избежать наиболее часто встречающихся ошибок. Но и после прочтения остается в силе следующий совет: как только вы почувствовали искушение написать собственный метод шифрования,

---

[1] Цит. по: «Ремни безопасности: страх оков» // Шпигель, 1975. — № 50 // www.spiegel.de/spiegel/print/d-41389557.html.

систему входа в учетную запись, интернет-магазин или систему оплаты кредитной картой, прервитесь, пожалуйста, уберите руки от клавиатуры и вышейте себе крестиком стенной коврик с надписью «Не делай сам!».

Иногда имеет смысл уже сам выбор языка программирования поставить в зависимость от аспектов безопасности. Если вы пишете программу, которой будут пользоваться другие, которая потенциально способна получить довольно широкое распространение и/или будет доступна напрямую через веб-интерфейс, то вам следует подумать как о злонамеренных, так и о небрежных пользователях. Для таких проектов ни в коем случае нельзя использовать язык C, а плохим программистам — в общем-то и C++. Проблема в близости к «железу» — в этих языках нет второго дна. Элементарные логические ошибки в управлении памятью очень легко могут вызвать совершенно непредвиденные эффекты: вы обращаетесь к элементу массива, которого не существует, что-то в него записываете, и в этот момент оказывается измененным адрес возврата функции, в которой вы сейчас находитесь. Выйдя из функции, вы попадаете в непредусмотренное место в коде, и если вам не повезло, то злонамеренный пользователь поместил туда код, который превратит ваш сервер в источник спама. В языках с автоматическим управлением памятью это невозможно.

Если вы используете SQL, то вам не помешает получше узнать о внедрении SQL-кода (SQL injection) (см. далее), если JavaScript — то о проблемах с XSS (см. далее).

Вам, как представителю целевой аудитории этой книги, надежнее будет использовать виртуальный хостинг, а не самостоятельно администрируемый сервер. Для профи все наоборот: на собственном сервере они, вероятно, установят более строгие стандарты безопасности, чем это сделал бы хостер. Но если вы не тратите значительную часть своего рабочего времени или досуга на администрирование сервера, то настройки по умолчанию и регулярное обновление системы защиты позволят даже со средненьким хостером добиться большей безопасности, чем вы смогли бы обеспечить сами.

Преимущества и недостатки открытости

Первая реакция, которая возникает у многих новичков при упоминании темы безопасности, это фраза: «Я держу свой исходный код закрытым, и поэтому никто не сможет найти в нем уязвимости». Этот подход, который по-английски называется также Security by Obscurity («Безопасность через неясность»), профессиональные разработчики отвергают, так как было замечено, что многими уязвимостями можно воспользоваться и тогда, когда исходный код программы неизвестен. К тому же представление о том, что слабые места кода никому не известны, искушает многих пренебрегать требованиями безопасности. Если вы раскрываете свой код, то тем самым вынуждаете себя вычистить уязвимости; кроме того, критика более опытных программистов может научить вас чему-то новому.

Поэтому с самого начала смотрите на свой код так, будто он открыт очам Господа Бога и всего мира (и дьявола, само собой). У этого есть дополнительное

преимущество: позже вы сможете пригласить в проект еще кого-нибудь и не стыдиться перед ним слишком сильно за качество кода.

«Никто ведь не знает URL» — плохой аргумент

То, что на URL нет ссылки, еще не является гарантией того, что до страницы никто не доберется. Назовем лишь одну возможность из многих: на секретной странице есть ссылки на другие страницы. Если вы щелкнете на одном из этих URL, адрес вашей страницы, к которой не ведут никакие ссылки, появится в чужих лог-файлах сервера и любопытные люди смогут его увидеть. Так как администраторы этих других сайтов также не слишком хорошо знают, что они делают, эти лог-файлы, чего доброго, еще и окажутся в Сети в открытом доступе и их можно будет найти в Google. И если хотя бы один из ваших пользователей установит в браузере панель инструментов Google или Alexa и воспользуется секретным URL, то довольно скоро мимо пройдет услужливый Google-бот и добавит ваши якобы неизвестные страницы к поисковому индексу. От этого вы еще можете защититься, изменив на своем сервере особый файл `robots.txt`. Но от того, что любой URL, к которому обратились иные лица, больше не будет секретным, вы защититься не сможете.

В случае ошибки возвращайте браузеру как можно меньше информации

Многие успешные атаки строились на том, что нападающие целенаправленно провоцировали ошибки и имели возможность получить из ответов системы информацию, нужную для успешных атак. Страницы с отчетом об ошибке, содержащие версию операционной системы, версию PHP или путь к кодовому файлу, вредны. Хотя законопослушным пользователям это и досадно, но с точки зрения безопасности лучше, чтобы страница просто выдавала «ошибку 500» и больше ничего. Если во время разработки кода вы в целях отладки активизировали функцию, которая подробно показывает вам все сообщения об ошибках, то перед реальным использованием вы должны ее отключить. Замените один из запросов базы данных в своем приложении на ошибочный и обратитесь к соответствующей странице, чтобы проверить, не передает ли ваш скрипт сообщение базы данных об ошибке прямо в браузер. Если логин или пароль были введены неверно, приложение не должно выдавать информацию о том, в какой части содержалась ошибка. Тогда исчезнет возможность с помощью скрипта отгадать имеющиеся имена пользователей.

Защитите каталоги от прямого доступа с помощью .htaccess

Обычно при создании ссылки на URL вместо example.com/some_directory/index.html пишут только example.com/some_directory/. Это срабатывает, потому что браузер самостоятельно ищет в данном каталоге файл с именем `index.html`. Если в каталоге нет `index.htm`., то может случиться так, что вместо этого любопытному посетителю будут представлены все файлы, находящиеся в этом каталоге, в виде списка. Это происходит, если на каком-либо сервере Apache в `httpd.conf-` или в ответственный за этот каталог файл `.htaccess` добавлена строка `+Indexes`. `Options -Indexes` запрещает отображать содержание каталога. Не рассчитывайте на то, что

более безопасная функция будет заботливо установлена по умолчанию. Часто это так, но, как показывает следующий пример, не всегда.

В 2008 году Heise Security сообщила о бреши в системе безопасности эротического концерна Узе Беате (heise.de/-202226), из-за которой в общий доступ попали 20 файлов, содержащих адреса электронной почты клиентов: «Сбой произошел вследствие целого ряда ошибок. Очевидно, администраторы сохранили данные для доступа к рождественскому онлайн-календарю с видеоклипами прямо на веб-сервере и к тому же активировали на сервере перечень файлов каталога. Таким образом, каждый интернет-пользователь мог ознакомиться с содержимым сервера просто с помощью браузера. Но этим дело не ограничивается: вероятно, Google-бот нашел ссылку на непубличные каталоги и недолго думая проиндексировал все данные, которые там нашел. В результате не только появилась возможность найти файлы случайно, но они еще и стали легкой мишенью для Google-хакинга»[1].

Будьте осторожны, пользуясь функцией CheckSpelling

Если эта функция активизирована в `httpd.conf-` или в каком-либо файле `.htaccess`, то посетителям несуществующей страницы или несуществующего каталога будут показаны альтернативные варианты с похожими именами. Там могут оказаться файлы, которые вы хотели держать в секрете.

Работа с паролями

В то время как имена пользователей обычно не являются тайной, пароли должны держаться в секрете не только от владельца сайта, но и от вас как администратора. Если постороннее лицо доберется до вашего файла с паролями, это может означать конец вашего веб-проекта, так как злоумышленник сможет нанести большой урон, всем честным пользователям потребуется сменить пароли и доверие по отношению к вам в целом будет разрушено.

Поэтому пароли ни в коем случае нельзя сохранять на сервере в виде незашифрованного текста. Ни в базе данных, ни как файл. Если кто-то посторонний получит доступ к месту, где хранятся пароли, то в его распоряжении в один миг окажутся счета всех пользователей. Уже тот факт, что вы как администратор интернет-ресурса имеете доступ к паролям в виде незашифрованного текста, будет восприниматься пользователями как нежелательный.

Потому при регистрации пользователя вам следует создавать хеш-код пароля (см. раздел «Хеш-коды, дайджесты, цифровые отпечатки» главы 26) и сохранять только его. Тогда хакер уже не сможет узнать пароль в виде незашифрованного текста, даже если взломает вашу базу данных с паролями. Этот хеш-код должен создаваться из пароля и случайной добавочной строки, потому что сейчас в Интернете можно найти списки хеш-кодов распространенных паролей. Если взломщику достаточно просто погуглить, чтобы узнать, какие пароли соответствуют хеш-кодам из вашей

---

[1] С помощью поисковых запросов вроде inurl можно использовать Google для поиска определенных уязвимых систем.

базы данных, то хеширование не добавит вашему проекту безопасности. Если же вы удлиняете введенный пользователем пароль, добавляя к нему какую-либо строку по собственному выбору, то эти списки вам не страшны. Этот прием называется «подсолка хеша» (hash salting), добавочную строку называют солью (salt). Строка для «подсолки» — это не тайное знание в области шифрования, а лишь способ сделать бесполезным поиск в списках хеша паролей. Поэтому вы можете спокойно хранить ее в той же базе данных, что и пароли, или поместить в файл конфигурации. Если она попадет в руки хакера, ничего страшного не случится.

Каждый раз, когда пользователь хочет зайти в свой аккаунт, создавайте из указанного пароля и «подсолочной» строки хеш-код и сравнивайте его с сохраненным. Если хеш-коды идентичны, то пароль, судя по всему, верный. Сейчас в качестве алгоритма для хеширования стоит использовать только bcrypt — все остальные слишком ненадежны.

Даже в больших компаниях, в которых работают опытные программисты, снова и снова случаются неприятные ситуации, связанные с хранением паролей. Редко происходит что-то более поганое, чем произошедшее с reddit.com, где в 2007 году украли резервную копию базы данных, в которой в незашифрованном виде хранились логины, пароли и адреса электронной почты всех пользователей. Такие истории столь многочисленны, что из них можно с уверенностью сделать один вывод: сохраняя пароли, вы, скорее всего, совершаете ошибку.

Методы аутентификации

Если только вы не разрабатываете самый простой статический веб-сайт с помощью клиента SFTP и текстового редактора, вам придется задуматься о том, кому и какие части сайта будет разрешено просматривать и кто будет иметь право изменять страницы, а это тут же повлечет за собой вопрос о том, с помощью каких методов аутентификации вы будете ограничивать доступ. По сути, вам нужно будет выбирать между следующими стратегиями аутентификации.

Вы полностью отказываетесь от аутентификации

Все пользователи имеют одинаковые права.

Преимущества таковы.

- Самое простое решение из всех возможных.
- Никаких огорчений из-за забытых паролей.
- Администраторы и пользователи точно знают, что их ждет, — никаких лживых обещаний обеспечить безопасность, хотя на самом деле ее нет.

Недостатки следующие.

- Вы должны быть способны быстро реагировать, чтобы суметь восстановить прежнее состояние в случае вандализма.
- Вы, будучи администратором, находитесь на одном уровне с пользователями. Например, у вас нет привилегированного доступа к собственным данным, постам и комментариям, или вам все же придется реализовать личную систему аутентификации для себя самого.

- Нельзя заблокировать пользователей, которые были замечены в чем-то нехорошем. (Впрочем, аутентификация также не является надежным решением этой проблемы. Можно быстро создать новый аккаунт.)
- Иногда запрашивается информация об IP-адресе, чтобы анонимных и неавторизованных пользователей все же можно было как-то идентифицировать (например, в истории правок страницы в «Википедии»). Такая идентификация с помощью IP-адресов — это лишь ориентировочная зацепка, так как один и тот же пользователь может иметь более одного IP-адреса и, наоборот, несколько пользователей могут приходить с одного и того же прокси-сервера и поэтому делить один IP-адрес.

HTTP-аутентификация (базовая аутентификация или дайджест-аутентификация)

Вы помещаете в файл `.htaccess` или в файл конфигурации сервера `httpd.conf` данные о том, какие пользователи имеют право доступа к тому или иному файлу или списку.

Преимущества таковы.

- Базовая аутентификация (Basic Authentication) — это наиболее простая форма аутентификации: никаких cookies, сессий, собственных страниц входа.
- Базовая аутентификация сравнительно просто конфигурируется. Возможно, ваш хостер даже предоставит вам в своем веб-интерфейсе для этого функцию вроде Создать защищенные списки, так что файлы `htaccess` или `httpd.conf` вам и видеть не придется.
- HTTP-аутентификация — это стандартизованная составная часть протокола HTTP. С ней может работать любой HTTP-клиент.
- HTTP-аутентификация является аутентификацией без сохранения состояния (stateless) (см. раздел «Состояние и отсутствие сохранения состояния» главы 26). Это значит, что каждый запрос будет обрабатываться независимо от того, что произошло раньше. Если остальное приложение работает без сохранения состояния, то имеет смысл сделать такой же и аутентификацию.

Недостатки следующие.

- Поля для ввода логина и пароля, как правило, выглядят неказисто, и исправить это нельзя. Нет возможности сразу перенаправить пользователя со страницы, где запрашивается пароль, к странице регистрации или к помощи по восстановлению забытого пароля.
- Регистрация новых пользователей становится не самой простой задачей.
- При базовой аутентификации пароль — если не применяется HTTPS — каждый раз передается в незашифрованном виде. Это еще хуже, чем сохранять его в незашифрованном виде, — это такая функция, которую никогда нельзя было изобретать. При дайджест-аутентификации передается только контрольная сумма пароля.
- Многие повышающие безопасность настройки, предусмотренные для дайджест-аутентификации, нужно устанавливать вручную.

- Отсутствует возможность автоматического выхода. Браузер помнит пароль по крайней мере до тех пор, пока не будет закрыта вкладка, а во многих случаях даже до тех пор, пока не будет закрыт браузер или пользователь вручную не очистит его историю. Если ваши пользователи посидят за чужим компьютером или ненадолго оставят свой собственный, это может стать проблемой: в этом случае любой неавторизованный пользователь хоть и не сможет запросто узнать пароль, но сможет пакостить на открытой странице.

Аутентификация на основе форм/сессии

На стандартной странице входа пользователей просят ввести логин и пароль. Если им удается войти, этот статус сохраняется на сервере в виде переменных сессии. Идентификационный номер, помещаемый в cookies или выдаваемый в виде URL, говорит серверу, какой пользователь к какой сессии относится.

Преимущества таковы.

- Страницу входа можно оформлять как угодно.
- Пользователи могут самостоятельно создавать новые аккаунты и восстанавливать забытые пароли.
- Сервер может установить лимит времени сессии и автоматически, например по прошествии определенного времени, «выставить» пользователя. Эти лимиты времени известны прежде всего по онлайн-банкингу — там продолжительность сессии сильно ограничена, обычно десятью минутами.

Недостатки следующие.

- Количество возможных ошибок, которые может сделать не слишком опытный программист, гораздо больше, чем при реализации описанных ранее методов. Ни шифрование, ни другие меры безопасности не являются технически необходимыми. Разработчик имеет все возможности для того, чтобы смастерить самое дырявое в мире решение для входа пользователей на сайт.
- Если cookie, в котором содержатся данные сессии, передается не через HTTPS, то возможен перехват сессии, что особенно актуально для открытых WLAN. Хотя у незваного гостя в этом случае и нет данных доступа, он остается на сайте до конца сессии и может успеть набедокурить.

Если вы решите использовать этот подход, то, пожалуйста, непременно воспользуйтесь готовыми решениями в виде отдельных библиотек или инструментов. Их ассортимент в настоящее время обновляется так быстро, что нет смысла советовать здесь что-то конкретное. Если вы зададите запрос authentication best solution в комбинации с названием нужного языка, то некоторые советы должны найтись. Обратите внимание на дату ответа. То, что год назад было оптимальным решением, сейчас, может быть, давно устарело.

OAuth/OpenID/Mozilla Persona

В этом случае вы полностью возлагаете задачу аутентификации на третью сторону. Для всех, кто не слишком хорошо знает, что делает, это одно из самых надежных

решений. Повсюду в Сети, где вы видите ссылки Войти через Facebook, Войти через Google или Войти через Twitter, разработчики сайта выбрали этот путь.

Преимущества следующие.

- Должно быть, у пользователей есть уже сотня разбросанных по Сети профилей и им не хочется создавать еще один специально для вашего приложения. Вместо этого они могут просто воспользоваться одним из существующих.
- У многих ваших пользователей большее доверие вызовут ресурсы, предоставляющие OAuth, то есть, например, Twitter, Google или Facebook, чем ваше собственноручно разработанное приложение GetRichQuick.
- Техподдержку в случае необходимости восстановить забытый пароль предоставляет кто-то другой. Вы экономите время и не так быстро теряете веру во вменяемость ближних.
- Данные, критические с точки зрения безопасности, сохраняются не на вашем сервере, а на серверах компаний, предоставляющих OAuth. Если хакер выкрадет все данные доступа, на вас ляжет лишь часть вины (за то, что вы доверились ненадежной третьей стороне). Это лучше, чем быть виноватым только самому. Кроме того, тогда вы сможете разделить свою досаду со многими другими людьми, которые положились на тот же ресурс.

Недостатки таковы.

- Некоторые пользователи будут *меньше доверять* таким компаниям, как Twitter, Google или Facebook, чем вашему сайту, особенно после того, как стало известно, что все крупные американские интернет-компании активно передают данные Агентству национальной безопасности. Если эта группа пользователей для вас важна, то вы можете использовать Mozilla Persona. В этом случае основная часть работы также будет сделана за вас, но Persona работает так, что Mozilla не получает информацию о пользователях.
- Пользователи, которые не разделяют этих опасений, но у которых просто нет аккаунта на одном из ресурсов, предоставляющих OAuth, будут вынуждены проходить муторный процесс регистрации на другом сайте, если только вы не предусмотрите для таких случаев собственную параллельную возможность аутентификации. Но вместе с этим исчезнут и многие преимущества решения с использованием OAuth.
- Некоторые пользователи не поверят, что ваше приложение GetRichQuick будет ответственно обращаться с их данными в Google и Facebook, зачастую очень личными.
- Если сайт, предоставляющий OAuth, в данный момент недоступен, то на ваш сайт никак нельзя зайти.
- Если вы не хотите полностью привязываться к определенному сервису OAuth, то можете предусмотреть несколько возможностей входа с помощью OAuth. Но тогда все резко усложнится.
- Для хакера хоть и гораздо сложнее, но и гораздо привлекательнее взломать масштабную, используемую по всему миру систему безопасности, а не ваше маленькое частное решение.

Внедрение SQL-кода и XSS — угрозы в пользовательском контенте

Одна из наиболее частых в нынешнее время проблем с безопасностью — это межсайтовый скриптинг (Cross Site Scripting, сокращается как XSS). JavaScript-сценарий из сомнительного источника подсовывается на какой-либо сайт и затем выполняется на браузере пользователя. Так может быть загружено любое содержимое, которое встраивается в страницу таким образом, что пользователь не распознает атаку. В худшем случае с помощью этого способа крадут данные кредитных карт, публикуют на чужих сайтах ссылки на спам или рассылают всевозможные сообщения с профиля пользователя в социальных сетях.

Угроза XSS крайне актуальна даже для больших ресурсов с профессиональным сопровождением. В начале 2013 года стало известно о вызванной XSS уязвимости на сайте Amazon, из-за которой посторонним стали доступны cookies сессии, а вместе с ними имена пользователей, адреса электронной почты, закупочные корзины и данные доступа. По сообщению Heise Security, «уязвимостью воспользовались тривиально. Нужно было лишь добавить на форум клиентов сообщение с заголовком, специально отформатированным примерно по такому образцу: `"><script>alert('XSS')<script>`. Так как Amazon плохо проверил заголовок добавленного сообщения, содержащийся в нем JavaScript-код был внедрен в определенные подстраницы форума и затем выполнен при обращении к нему браузера». За пару месяцев до этого система онлайн-платежей Paypal также отличилась XSS-уязвимостью «в ключевом месте». В 2011 году Heise Security сообщила о XSS-уязвимостях на сайтах 17 банков — проблемы были обнаружены школьниками.

Вторая основная угроза, связанная с пользовательским контентом, — это атаки, в ходе которых с помощью так называемого внедрения SQL-кода (SQL injection) считываются и изменяются ваши базы данных. Если такое случится с вами, это будет не просто досадно — вы потеряете доверие пользователей. Данные вы сможете восстановить с помощью хорошей резервной копии, а вот вернуть доверие удастся едва ли. Если, программируя, вы не ставите во главу угла осторожность, то для такой атаки будет достаточно уже того, что ваше приложение снабжено формой запроса и управляет данными в БД.

У обеих проблем общие корни: у пользователя есть возможность внедрить в вашу систему активное содержимое с помощью обычных текстовых полей, параметров URL-адреса и cookies. Затем оно выполняется в системе (внедрение SQL-кода) или, непроверенное, передается браузеру и выполняется им (XSS-атака). Чтобы воспрепятствовать этому, нужно изначально рассматривать все данные, поступающие в ваше приложение из внешней среды, как потенциально опасные. Они должны быть очищены от управляющих символов, которые в них, возможно, содержатся. Это касается не только тех данных, которые пользователи могут ввести в специально предусмотренные для этого поля, но и параметров URL-адреса, cookies, поля User Agent браузера, данных из БД, файлов конфигурации и файловых систем, RSS-каналов и даже штрихкодов. Подозрительно все, что приходит от программ, которые написали не вы сами. «До использования»

означает в данном случае также «до того, как сохранять данные в БД или на жесткий диск, чтобы прочитать их в будущем».

В библиотеках есть функции для очистки входных данных, которые различаются в зависимости от языка и используемой библиотеки. Перед записью текста в базу данных нужно очистить его от фрагментов SQL-кода, которые, возможно, были в него внедрены. Для этого можно воспользоваться параметризованными запросами (см. далее) или SQL-фреймворком. Внедрение JavaScript-сценариев предотвращается за счет того, что в тот момент, когда сгенерированное пользователем содержимое попадает в систему, оно проверяется по белому списку (также см. далее) и опасные символы удаляются.

Внедрению SQL-кода можно воспрепятствовать с помощью установки фильтров на входе, так как SQL-код выполняется прямо на сервере. Но, учитывая угрозу XSS-атак, вы в любом случае должны позаботиться о том, чтобы фильтровались также выходные данные, которые ваша система передает браузеру. Здесь основная суть в том, чтобы обезвреживать управляющие последовательности, которые могут содержаться в HTML-тегах и JavaScript-сценариях.

Проверка должна происходить на стороне сервера

Если вы проверяете пользовательский контент только с помощью JavaScript на компьютере пользователя, то с таким же успехом можете вообще ничего не делать. Программист полностью контролирует только сервер и запущенные на нем программы. Браузер работает на компьютере пользователя и может быть заменен скриптом. Данные от компьютера пользователя к серверу передаются через множество промежуточных пунктов. Каждый из этих роутеров также может считывать и изменять данные. Всякий, кто готов потратить две минуты на поиски инструкции, может как угодно манипулировать данными проверки, выполненной на стороне клиента. Стоит выполнить проверку на стороне клиента, а затем сообщить пользователю, что с введенными им данными или загруженными фото что-то не в порядке. Но это лишь делает приложение более дружественным по отношению к пользователю, а с точки зрения технической безопасности проверка бесполезна. Вам в любом случае придется повторить проверку на стороне сервера.

Данные, введенные пользователем, не следует сразу же отправлять на выход

Вам не стоит просто так выводить данные, которые ввел пользователь, не выполнив перед этим экранирования проблемных символов. В каждом языке для этого есть готовые команды. В PHP для SQL используют `mysql_real_escape_string`, а для данных, которые выводятся на веб-сайте и, возможно, содержат JavaScript-сценарии, — `htmlspecialchars`. Для Shell-команд есть `escapeshellcmd` и `escapeshellarg`. Все эти функции снабжают определенные знаки обратной косой чертой (\) и таким образом не допускают выполнения «контрабандного» кода, который, возможно, содержится в данных.

Поэтому не пишите:

```
echo ,Вы ввели , . $_GET["UserInput"];
```

Лучше вставить данные, введенные пользователем, в переменную, превратить некоторые проблемные знаки в HTML-код и только затем вывести их. В нашем примере на PHP это выглядело бы так:

```
$rawUserInput = $_GET["UserInput"];
$userInput = htmlspecialchars($rawUserInput);
echo 'Вы ввели , . $userInput);
```

`htmlspecialchars` преобразует знаки `&`, `<`, `>`, `"` и `'` в их HTML-эквиваленты. Если введенный текст содержит разметку HTML, то на выходе вместо `<script>` будет стоять безобидное `&lt;script&gt;`. Для честных пользователей, которые просто хотят добавить в свое сообщение пример с HTML, этого будет достаточно, но потенциально вредоносный JavaScript-сценарий теперь не будет выполнен.

Конечно, `htmlspecialchars` не защитит от всех угроз. Если ваши пользователи больше обычного склонны напарываться на хакерские атаки или речь идет об очень критических данных, из-за чего вам необходимо применять более высокие стандарты безопасности, то этой стратегии будет недостаточно. Далее, в разделе «Белые списки лучше черных», мы поговорим об этом подробнее.

Не применяйте SQL, JavaScript и Shell-команды напрямую для выдачи данных

Не составляйте запросов или команд из данных, введенных пользователем, не выполнив перед этим экранирование проблемных символов.

Не обращайтесь к приложениям, имена которых взяты из данных пользователей

Не запускайте с помощью `system()`, `eval()` или похожих обращений локальные программы, если на имя или путь программы могли повлиять извне. Про `eval()` лучше вообще забыть, а `system()` используйте очень осторожно.

Проверяйте вводимые данные на достоверность

Загружаемые данные должны иметь ожидаемый формат, а не какой угодно. Это же касается переменных, с которыми вы затем планируете работать. В типобезопасных языках (см. главу 26) вы не сможете не заметить, что переданная пользователем переменная `id` на самом деле содержит не число, а половину романа. Если же вы пишете на языке со слабой типизацией, вроде JavaScript или PHP, вам придется самому позаботиться о том, чтобы переменная была ожидаемого типа. В PHP с помощью `is_int()`, `is_numeric()` и т. д. вы можете выяснить, что перед вами. С помощью `intval()`, `strval()`, `boolval()`, `floatval()` или `settype()` можете принудительно задать тип переменной. В JavaScript можно воспользоваться `parseInt()`, `parseFloat()` и `isNaN()` (is not a number). Кроме того, `filter_var` в PHP дает различные возможности для того, чтобы выяснить, соответствует ли переменная определенной схеме — например, схеме правильного адреса электронной почты или URL. Многочисленные готовые фильтры отлично заменяют, наверное, около 99 % того, что у вас возникает искушение написать самому (см. также главу 19).

Если у вас в back-end база данных, используйте параметризованные запросы. Если у вас есть система управления контентом, которая использует URL вида /mypage?id=53 и эти ID являются первичными ключами в базе данных, то лучше работать с помощью параметризованных запросов

Ни в коем случае не пишите в вашей программе SQL-строки, просто вставляя содержимое, приходящее извне. То есть никогда не пишите вот так:

```
var query = "select * from mytable where id = "+$pageId;
var result = query.execute();
```

Параметризованные запросы в разных языках выглядят по-разному, но работают примерно так:

```
var query = "select * from mytable where id = ?";
query.setParameter(1, $pageId);
var result = query.execute();
```

При замене вопросительного знака содержание `$pageId` будет проверено на наличие фрагментов SQL-кода и при необходимости обезврежено.

Проверяйте размер загружаемых файлов

Не рассчитывайте на то, что никому в голову не придут странные идеи и выделенного места точно хватит, — в этом случае вы открываете свою систему для DoS-атаки, основанной на нехватке файлового пространства (file space denial of service attack), в ходе которой пользователи загружают коварно устроенный ZIP-файл, съедающий все свободное место на жестком диске вашего сервера. Это так называемые архивные бомбы — очень маленькие (около 40 Кбайт) файлы, которые при распаковывании превращаются в файлы объемом тера- или даже петабайт. Они используют тот факт, что алгоритмы сжатия могут невероятно сильно сжимать абсолютно идентичные файлы, например содержащие только число 0. В качестве контрмеры вам стоит не просто распаковывать загруженные архивы вслепую, а сначала проверять их размер с помощью libzip.

Не помещайте SQL-запросы в URL в качестве строки запроса

Поиск в Google по запросам `inurl:&` `inurl:select` `inurl:from` `inurl: where` извлекает на свет божий удручающее множество результатов. Даже люди без хакерских наклонностей и способностей при виде такого едва ли смогут удержаться и не совершить — с помощью простого изменения URL — прогулку по чужой базе данных.

Не помещайте в URL ничего другого, что может быть критическим с точки зрения безопасности

Разработчикам систем интернет-магазинов необходимо решить следующую проблему: каждый покупатель должен иметь возможность заполнять корзину товарами, затем перемещаться по интернет-магазину дальше и в конце концов пройти в кассу, чтобы оформить заказ. Так как HTML является протоколом без сохранения состояния (см. раздел «Состояние и отсутствие сохранения состояния» главы 26),

разработчик должен задуматься о том, каким образом интернет-магазин будет запоминать состояние корзины и соотносить его с определенным покупателем.

Мы рекомендуем решение, заключающееся в том, чтобы воспользоваться веб-фреймворком, который будет с помощью cookies и параметров URL открывать сессию для каждого пользователя и в этой сессии сохранять состояние корзины. Компетентные решения этой задачи существуют для большинства языков.

Некоторым не слишком опытным разработчикам однажды пришла в голову идея кодировать артикульные номера позиций корзины, а также их цены в URL или в cookies сессии. Если пользователь переходил на другую страницу, то данные о корзине передавались от браузера клиента серверу, а от сервера — с новой страницей — снова браузеру. В этом случае искушенные покупатели могли изменить количество артикулов и цену и купить не три утюга по 29,9 евро каждый, а пять утюгов по 2,99 евро.

Ошибка заключалась в доверии к данным, которые вернулись от браузера пользователя. Сервер должен был только выдавать данные о цене, но никогда не должен был принимать их обратно. Окончательный расчет следовало производить исключительно на сервере, единственные данные, которые можно было принимать от браузера, — это количество товаров и артикульные номера.

Поэтому никогда не стоит передавать пароли, номера кредитных карт, цены и подобные данные на сторону клиента, а потом снова принимать их от клиента на сторону сервера. И пожалуйста, если вы собираетесь иметь дело с кредитными картами пользователей, раздобудьте достаточно денег, чтобы нанять эксперта по безопасности.

Не выдавайте слишком много внутренних секретов вашего приложения

Снаружи должно быть как можно меньше видно, как вы храните свои данные, — не нужно создавать лишние уязвимые места.

В 2009 году журнал «Шпигель» сообщал: «В книжном интернет-магазине Libri.de произошел серьезный сбой. Просмотр 563 640 счетов сотен тысяч покупателей стал доступен любому интернет-пользователю». Для скачивании счета за покупку в виде PDF-документа вы получали порядковый номер. Вводя другие случайно выбранные номера, можно было скачать все счета других покупателей за прошедшие 16 месяцев. «Шпигель» иронизировал: «Особенно неловкое обстоятельство: на Libri красуется знак качества экспертной организации TÜV Süd за обеспечение безопасных покупок в Интернете».

Здесь совместились две проблемы с безопасностью. Существенной уязвимостью было отсутствие проверки на наличие у пользователя права доступа к определенному счету. При каждой попытке получения доступа к счету нужно проверять, есть ли у пользователя права доступа для этого счета. Однако, хотя это уже само по себе было грубой ошибкой, ничего плохого, возможно, все же не произошло бы, если бы в системе безопасности не было второй бреши: в веб-приложении применялись внутренние идентификационные номера, вероятно, из таблицы базы данных. Атакующим было легче легкого угадать порядковые номера. Небольшой дополнительный шаг сделал бы эту атаку невозможной — например, можно было бы добавить к порядковому идентификационному номеру «соль» (см. ранее) и за-

тем перевести его с помощью алгоритма хеширования в хеш-сумму. На стороне сервера было бы очень легко, создав запись в базе данных, связать имя пользователя, номер счета и хеш-сумму, и тогда на стороне клиента стало бы невозможно отгадать URL для чужого номера счета.

Если в языке, на котором вы пишете, есть функция taint checking (проверка на уязвимости), воспользуйтесь ею

В Ruby и Perl `taint` относится к стандартным функциям, во многих других языках ее можно подключить через библиотеки или расширения (поиск по таким запросам, как taint php или taint python, приведет вас к нужному решению). В `taint mode` программа создает список переменных, на которые может быть оказано воздействие извне, а также переменных, производных от них. В дальнейшем все эти переменные считаются ненадежными. При попытке применить их в опасных местах, например в SQL-командах или командах на уровне операционной системы, таких как `eval()`, их выполнение прерывается.

Белые списки лучше черных

Программа, с помощью которой реализован ваш сайт, — это посредник, выполняющий действия пользователей. Чтобы ограничить действия пользователей, нужно ограничить возможности программы выполнять те или иные действия. Добиться уверенности в том, что программа будет делать только то, что должна, можно двумя способами: или вы запрещаете делать то, чего нельзя делать, или разрешаете делать только то, что можно. Этот основополагающий принцип встречается в различных формах в самых разных местах. Например, можно отфильтровывать из данных, вводимых пользователем, все теги `<script>`, а остальные теги оставлять неизменными (это называется созданием черного списка), а можно удалять все теги, но разрешить пару разметочных вроде `<b>` и `<i>` (создание белого списка).

С точки зрения безопасности всегда имеет смысл разрешать как можно меньше, то есть сначала все запретить и разрешать только то, что наверняка не причинит вреда. В нашем примере сохраняется еще много возможностей внедрить в сайт JavaScript-сценарий, кроме тегов `<script>`, например, это можно сделать с помощью `<p onmouseout="">`. Проблема будет надежно устранена только в том случае, если будут отсеиваться вообще любые теги, кроме тех, которые указаны в списке разрешенных. Или можно вообще запретить HTML-разметку и заменить ее на Markdown. Тогда жирный шрифт будет отмечаться с помощью `**b**`, а не `<b>`. При создании белого списка у вас как у программиста будет больше работы: сначала придется выдумывать белый список, а потом дополнять его кучей вещей, которые вы забыли внести в него с самого начала. С этим недостатком ничего нельзя поделать. Неполнота белого списка будет вновь и вновь раздражать вас, однако это раздражение в целом меньше, чем потенциальные беды, которые угрожают вам в ином случае.

Избегайте регулярных выражений вроде `.*`. Избегайте также non-greedy-варианта `.*?`. Короче, не говорите вашему регулярному выражению: «Используй

все, что стоит на этом месте, — оно уж точно сгодится!» Лучше определить символьные классы с отрицанием, то есть, например, `[^"]*` — «используй все, пока не появится кавычка». Хотя это все еще и черный список, но уже прогресс. Однако все же лучше положится на белые списки — точно задать, какие символы можно принимать, например `[a-zдць0-9]*`.

Установите все ползунки на минимум

Принцип белых списков можно обобщить, и тогда он будет звучать так: все элементы в вашем коде и вокруг него должны получать только самые необходимые права. «Все запретить» — плохой принцип при воспитании детей, однако в деле разработки ПО зарекомендовал себя хорошо. Все элементы, объекты, функции и переменные вашего кода не должны ни с кем связываться, не должны быть ни для кого доступными, не должны принимать от незнакомых дядь печенюшки, только если вы явно не разрешили это. Подумайте также о том, что произойдет, если окажется, что вы все же не заметили какую-то уязвимость: не станут ли в таком случае доступны атакующему и все остальные части системы?

Держите ресурсы, важные с точки зрения безопасности, раздельно

Если база данных является частью интерактивного веб-сайта, то веб-сервер и исходный код должны быть по возможности запущены на другой машине, а не на той, где находится база данных. Если эта чужая машина будет скомпрометирована, то у непрошеных гостей все еще не будет полного доступа к серверу с базой данных.

При конфигурировании большинства реляционных баз данных вы можете организовать допуск разных пользователей по одной схеме. Например, у вас может быть пользователь `admin`, который осуществляет вход с `localhost`, то есть с машины, на которой находится база данных. Эта учетная запись защищена дважды: нужно знать пароль администратора, а также предварительно зайти на сервер с базами данных. Поэтому у данного пользователя есть все права: он может создавать и удалять (`drop`) базы данных, создавать учетные записи и изменять права пользователей. Так как у такого пользователя-администратора столь широкие полномочия, то они должны использоваться действительно только для администрирования баз данных, а не для того, чтобы, например, администрировать интернет-магазин.

Для этого есть пользователь с ограниченными правами, например, с именем `webshop`. Он связывается с сервера с базой данных, чтобы читать товарные позиции и добавлять заказы. Как того и требует здравый смысл, у этой учетной записи нет локального логина на сервере с базами данных, а вместо этого она соединена с базой данных напрямую. Сразу максимально лимитируется число IP-адресов, с которых из учетной записи `webshop` можно соединяться с базой данных, — оно ограничивается веб-сервером и компьютером администратора баз данных и/или разработчика. Права этой учетной записи сильно ограничены: нельзя создавать, изменять и удалять базы данных, таблицы, учетные записи, а также изменять права пользователей. Можно только читать, изменять и удалять строки таблиц.

Если в нашем воображаемом интернет-магазине есть еще одно отдельное приложение, которое отвечает за систему управления товарами, то ему можно поставить в соответствие еще одну учетную запись, из которой можно будет создавать и удалять товары, изменять цены и вводить скидки. В таком случае пользователь `webshop` имеет доступ к схеме управления товарами с правом только на чтение.

С помощью такой ступенчатой системы прав доступа вы можете минимизировать последствия, если реализуется наиболее вероятный сценарий: кто-то взломает веб-сервер и сможет с правами интернет-магазина соединиться с базой данных. Так как он не сможет манипулировать товарами и ценами, ему будет гораздо сложнее сделать крупный заказ дорогих товаров по смешной цене. И он не сможет запросто удалить или изменить вашу базу данных, так как эти права есть только у администратора. В худшем случае он сможет просмотреть и изменить поступившие заказы, что уже довольно неприятно, но не сможет, скомпрометировав сервер, получить контроль над остальной системой.

Как можно сильнее ограничивайте права файлов и каталогов

При установке чужих инструментов и скриптов на ваш сервер сначала часто ничего не работает, так как определенные файлы или каталоги не обладают достаточными правами. Конечно, проще всего решить эту неприятную проблему одним махом, установив все права на «777» (все могут читать, писать и выполнять файл). Но все же не делайте этого. «777» вообще не следует использовать, так как в этом случае какой-нибудь скрипт может написать и выполнить в файле `rm -rf *` — и плакали ваши данные. Хотя сценарии реальных атак и сложнее, но комбинация из «каждый может делать записи в файле» и «каждый может выполнять этот файл» взрывоопасна.

Другие люди, которые имеют доступ к приложению, данным, серверу или локальному компьютеру, должны иметь как можно более ограниченные права

Вашим друзьям и сослуживцам вовсе не нужно иметь недобрые намерения, чтобы натворить нехороших дел. Хватит халатности, рассеянности или незнания. Вот пример пользователя blowdart, приведенный в подборке Stack Overflow *Worst security hole you've seen* («Худшая уязвимость, с которой вы сталкивались»): «Много лет назад моя фирма захотела улучшить поиск по своему ASP-сайту. Ну, я поставил Index Server[1], исключил пару каталогов admin из индексации, и все было хорошо. Но кто-то без моего ведома предоставил сотруднику отдела продаж FTP-доступ к веб-серверу, чтобы он мог работать на дому. Это было еще во времена коммутируемого доступа, и для него это был самый простой способ обмениваться файлами. Ну и он начал загружать на сервер разные штуки, среди прочего документы, в которых были расписаны наши надбавки на все услуги ...и Index Server все проиндексировал и выдавал это в результатах, когда пользователи производили поиск по запросу "цены"» (stackoverflow.com/questions/1469899/worst-security-hole-youve-seen).

---

[1] Index Server — это ныне устаревший сервис Windows, с помощью которого можно было производить поиск по локальным компьютерам, внутренним сетям и веб-сайтам.

Не игнорируйте предупреждения компилятора

Хотя они и не имеют непосредственного отношения к безопасности, однако игнорирование маленьких проблем часто вызывает большие. Кроме того, предупреждения компилятора все же могут иметь отношение к безопасности, просто вы пока не разглядели взаимосвязь.

Черный ход тоже запирайте

Разработчики очень редко разграничивают различные компоненты своего веб-приложения. В части случаев это и невозможно, к примеру, потому, что имя или пароль пользователя базы данных должны быть указаны в коде или файле конфигурации. Если постороннему удается получить доступ к одному компоненту, то перед ним обычно довольно легко открывается путь и к другим компонентам, таким как база данных или интерфейс администратора. Поэтому решение проблемы безопасности всегда надежно лишь настолько, насколько надежен самый уязвимый компонент. Тот, кто однажды прикрепил свой велосипед дорогим замком к хлипкой деревянной изгороди, а потом был вынужден возвращаться домой пешком, уж точно запомнит это правило.

Если вы полностью сконцентрируетесь на защите от внедрения SQL-кода и межсайтового скриптинга, то можете легко упустить из виду тот факт, что возможными входными воротами для злоумышленников является не только HTML-front-end.

Классическая ошибка, которую особенно часто совершают при разработке веб-приложений, заключается в том, что, хотя работа приложений и их защищенность от внешних угроз протестированы и являются относительно надежными, их данные находятся в общем доступе Windows и доступны для некоторых рабочих групп или вообще для всей фирмы. Или административная страница сервера защищена более слабым паролем, чем веб-приложение.

Известным примером является phpMyAdmin — инструмент администрирования баз данных на веб-основе для БД MySQL. За годы использования программы в ней было обнаружено более 130 уязвимостей — не все они существенные, но многие дают возможность получить несанкционированный доступ к серверу, на котором установлен phpMyAdmin. Тот, кто устанавливал этот инструмент, для того чтобы ему удобнее было создавать свою базу данных, потом часто забывал удалить или обновить его. Между тем появились скрипты, которые автоматически ищут устаревшие версии phpMyAdmin на чужих серверах и пытаются использовать их уязвимости. Неважно, насколько хорошо защищено ваше собственное веб-приложение: если на том же самом сервере установлена ненадежная версия phpMyAdmin, то сервер, а вместе с ним и ваше приложение подвержены угрозам.

Не встраивайте незащищенный черный ход «на минутку» для тестирования. Впервые описанная Джоном Голлом теорема Роэ гласит: «Разработчики систем любят создавать методы, позволяющие им самим обходить систему». Это значит не только то, что политики в социалистических странах любят окружать себя роскошью. Это значит также и то, что у вас появится искушение создать себе аккаунт администратора со всевозможными привилегиями и прошитым паролем. Не де-

лайте этого. Даже в качестве временного решения. Иначе вас постигнет та же участь, что и Telekom в 2008 году, когда «Шпигель» выяснил, что данные более чем 30 млн клиентов T-Mobile оказались открытыми для просмотра и изменения с любого компьютера. Виной тому был один-единственный пароль администратора, который знали не только бесчисленные работники магазинов T-Punkt, но и разнообразные хакеры, а под конец и редакция «Шпигель».

Если вы считаете необходимым соблюдать определенный уровень безопасности, то вам придется мириться с определенными неудобствами. Например, пароль от привилегированного аккаунта должен быть особенно надежным, система должна заставлять менять его каждые пару недель, и доступ к этому аккаунту администратора должен быть доступен только через определенные IP-адреса — лучше всего, чтобы они принадлежали к одной внутренней сети. Правда, это может, увы, привести к тому, что вам придется в воскресенье ехать в фирму, чтобы снова запустить систему.

Тестирование на проникновение

Существуют так называемые комплекты инструментов для тестирования на проникновение (Penetration Testing Toolkits) — программы, с помощью которых вы можете проверить свое веб-приложение на наличие явных уязвимостей. К сожалению, дела с ними обстоят так же, как и с шифрованием: они требуют компетентности и знания сценариев атак, от которых вы хотите защитить веб-приложение. Неполное знание может оказаться опасным, особенно в соединении с переоценкой собственных сил.

Автоматизированные инструменты для тестирования на проникновение никогда не могут протестировать все возможные сценарии атак, чтобы они были по-настоящему эффективными, их нужно настроить на веб-сайт, который будет подвергаться тестированию. Компетентный тестировщик с помощью такого инструмента может протестировать широкий спектр аспектов безопасности, но это ни в коем случае не панацея.

Поэтому мы рекомендуем использовать эти инструменты лишь при условии, что вы осознаете, что это полумера: если вы применяете один из них, то не должны впасть в заблуждение и считать, что ваш сайт надежно защищен, раз инструмент не нашел ни одной уязвимости. В наших несистематических тестах эти инструменты наряду с правильными давали как ложноположительные (нахождение уязвимостей, когда их на самом деле нет), так и ложноотрицательные (возможные проблемы, которые инструмент не фиксирует) результаты.

Один из примеров инструмента для тестирования на проникновение — Vega. Этот инструмент запускается на вашем компьютере и проходит по ссылкам на ваш сайт, надеясь в какой-то момент случайно наткнуться на ненадежную форму. Он представляет результаты в виде симпатичного списка. Еще один, менее удобный инструмент — OWASP ZAP. Как OWASP ZAP, так и Vega могут вклиниться в качестве прокси-программы между вашим браузером и сервером и анализировать трафик, в то время как вы перемещаетесь по сайту. Чаще всего это более перспективный образ действий, чем проверка в полностью автоматическом режиме, выполняемая Vega.

Ошибки других

Если теперь, после прочтения этой главы, вы знаете о безопасности чуть больше, чем раньше, мы надеемся, что это не приведет к тому, что вы начнете отпускать в Интернете ехидные замечания по поводу уязвимостей, обнаруженных в чужих крупных проектах. Разница не в том, что у других эти уязвимости есть, а у вас их нет, а в том, что у других они уже вскрылись, а у вас — еще нет. Если вы обнаруживаете на чужом сайте или в чужом коде проблему, не думайте: «Хо-хо, они знают еще меньше, чем я!» Или ладно, если очень уж хочется, думайте. Но тогда будьте столь любезны и напишите дружественное электронное письмо.

Как это делается, можно увидеть на примере сообщения от Хельги Гримхардт, которое пришло в Riesenmaschine в 2008 году. Поскольку многое в нем достойно подражания, мы дополнительно разъяснили все важные элементы в сносках.

> Уважаемые дамы и господа!
>
> Недавно я заглянула на riesenmaschine.de. При этом я заметила на сайте некоторые уязвимые места, которые позволяют внедрить на сайт инородное содержимое. Например, атакующий мог бы запросить данные пользователя, а пользователь не заметил бы, что запрос поступил не от riesenmaschine.de.
>
> Речь идет об уязвимости, связанной с так называемым межсайтовым скриптингом (XSS)[1]. Следующий пример[2] показывает, как через нормальную ссылку может быть внедрено чужое содержимое (в данном случае — Iframe на Google).
>
> http://riesenmaschine.de/tshirts2006.html?shirt=1%22%3E%3Ciframe%20src=http://www.google.de/%3E
>
> Естественно, теоретически можно также подключить и другое менее приятное содержимое (теги, скрипты и т. д.), однако этот пример очень наглядно демонстрирует принцип и при этом не опасен. Проблему можно сравнительно легко устранить с помощью фильтрации входящих данных[3].
>
> Я ни в коем случае не хочу напугать вас[4] (я знаю людей, которые отвечают на указания такого рода угрозами), но думаю, что безопасность сайтов можно повысить только таким способом, и поэтому стараюсь указывать разработчикам на уязвимости, которые бросились мне в глаза.

---

1 Автор не просто написала «XSS-уязвимость», а сперва описала проблему, а потом указала ее наименование.

2 Пример никогда не помешает.

3 Указание на способ решения проблемы, снабженное успокаивающим замечанием, что это решение не дико сложное. Если хотите сделать еще лучше, присоедините ссылку на инструкцию.

4 Автор ставит себя на место адресата и исключает возможные опасения.

Если вы захотите еще о чем-то спросить, то я, конечно, с радостью вам отвечу[1].

С уважением, Хельга Гримхардт

Безопасность — это процесс

Избавьтесь от мысли, что безопасность можно добавить к коду в качестве последнего штриха. Безопасность начинается, как только вы начинаете размышлять о том, как будет работать программа. В тот момент, когда вы пишете первые строки, уже закладываются основы будущей безопасности. Вопросы безопасности проявляются не только тогда, когда пользователь получает предупреждение, что для совершения действия ему нужно осуществить вход, — они пронизывают весь код целиком. Но безопасность не ограничивается только кодом. Безопасность — это широкая концепция, которая наряду с исходным кодом должна охватывать и другие аспекты работы организации, даже если организация состоит только из вас самих. Сюда относятся доступ к серверу, как физический, так и удаленный, пароли к базам данных и, начиная с определенного уровня важности данных, также выбор людей, которые имеют доступ к данным. Не давайте обещаний обеспечить безопасность, если множество ваших коллег могут считывать данные из БД и сохранять на флешку. В области безопасности не бывает «довольно хороших» решений, когда существование фирмы, университетского проекта или развлекательного интернет-сообщества может оказаться под угрозой из-за нарушения правил безопасности. В таких случаях спасет только помощь экспертов.

Если есть хоть какая-то возможность, то вы должны оказывать противодействие, когда сослуживцы или заказчики пренебрегают концепцией безопасности как процесса и при этом ожидают вашей помощи. Не перекладывайте ответственность на плечи других сотрудников («это работа сисадминов») или конечных пользователей кода («они так захотели»). С безопасностью в IT-сфере дела обстоят так же, как с гигиеной в больницах: что-то может получиться только в том случае, если все вовлечены в процесс и всем, кому нужно, постоянно и терпеливо напоминают о правилах. Вы не освобождены от работы над вопросами безопасности только потому, что вы не слишком хороший программист. Как только вы принимаете решение писать код, который будет использоваться не только вами, вы берете на себя ответственность за данные других людей или по меньшей мере за безопасность компьютера, на котором запускается ваша программа или открывается ваш сайт. Когда речь идет о дорожном движении, то даже в качестве пешехода вы несете определенную ответственность за безопасность других людей, пускай и ограниченную. Например, другие участники дорожного движения рассчитывают, что вы не броситесь ни с того ни с сего под машину. Будучи программистом, который имеет дело с задачами, связанными с безопасностью, вы уже не пешеход, а как минимум велосипедист. Вы должны осознавать эту ответственность.

---

[1] Роскошный бонус — предложить дальнейшую помощь.

Каждый человек может иметь дело лишь с ограниченным количеством угроз, остальные вытесняются с целью сохранения душевного здоровья. Если вы чувствуете, что борьба с повседневными рисками и так отбирает у вас все силы, то вам необходимо расставить приоритеты. Лучше регулярно делайте пенсионные отчисления, поскольку старость приближается неумолимо, тогда как написание кода, связанного с аспектами безопасности, — дело полностью добровольное. Просто оставьте его другим людям. Есть много областей, от которых не слишком опытным программистам лучше держаться подальше. Способность видеть и учитывать границы своей компетентности и своих возможностей делает вас не более плохим, а более хорошим программистом.

26 Полезные концепции

> Скандал: только в 33 года я узнала, что олень — это не самец косули! Система образования, государство, родители — никто не справился со своей задачей!
>
> *Frollein/@dorfpunk, Twitter, 17 мая 2011 года*

Эта глава являет собой позитивную противоположность главе 17 «Предупреждающие знаки». Хотя информационные технологии развивались очень быстро и по сравнению с техническими дисциплинами это область относительно молодая, однако существует несколько концепций, которые хорошо себя зарекомендовали и поэтому в неизменной или мало меняющейся форме снова и снова появляются в деле разработки ПО.

Хорошо иметь некоторое представление о перечисляемых в этой главе идеях и практических методах и даже кое-когда применять их, но и в противном случае вас не поразит гром с небес. Самое плохое, что с вами может произойти, — это то, что жизнь будет чуть более трудной и вы не сможете понять пару кусков чужого кода.

Исключения

Исключения (exceptions) — это механизмы, позволяющие вывести ошибки, встречающиеся в глубине программы, на более высокие уровни — в направлении пользовательского интерфейса или программного управления. Часто разработчики сталкиваются со следующей проблемой: они обращаются к функциям, чтобы, допустим, установить связь с базой данных, эти функции, в свою очередь, обращаются к другим функциям, и может случиться так, что ошибка, например обрыв соединения с БД, произойдет в какой-нибудь функции библиотеки, в то время как функция, которая хотела воспользоваться соединением с БД, находится десятью уровнями выше. Если функция A обращается к функции B, то устанавливается прямое соединение: A может передавать B значения через параметры функции, а B может возвращать значение A. Если же функция C обращается к функции A, а A обращается к функции B, то между B и C уже нет прямой связи.

Если функция не знает, откуда к ней обратились изначально, то она не может отправить туда сообщение об ошибке. Возможно, было правильно принудительно

завершить программу из этой глубоко расположенной функции (если дальнейшая работа не имеет смысла), но часто это не входит в планы программиста, который хотел бы выдать пользователю сообщение об ошибке, спокойно закрыть файлы или сообщить об ошибке другим системам.

Чтобы закрыть пробел между появлением ошибки в одной функции и ее обработкой совсем в другой, раньше иногда работали с кодом ошибки (error code): каждая функция отправляла назад код ошибки или 0 при отсутствии ошибки и должна была следить за тем, чтобы коды ошибки каждой вызываемой функции она сама выдавала как код ошибки. Это было утомительно, и такая система была подвержена появлению ошибок, поэтому уже в 1960-х годах начались попытки создать другой, более глобальный канал коммуникации на случай возникновения ошибок, из которого позже развились исключения и обработка исключений (exception handling)[1].

Коммуникация между частями программы или между библиотеками и программой, обращающейся к ним с помощью исключений, происходит следующим образом: вышестоящая функция дает знать, что она интересуется исключениями в коде, к которому обратилась (она хочет поймать исключения). Библиотечная функция, в свою очередь, в форме исключения дает знать, что возникла серьезная ошибка (бросает исключение). Программа останавливается, и обработчик исключений ищет среди вышестоящих функций первую функцию, которая может ловить исключения. Если обработчик исключений не находит ни одной такой функции, программа завершается. Если он находит такую функцию, то программа продолжается с нее и дает ей возможность обработать исключение и принять надлежащие меры. Это выглядит примерно так:

```
function openDatabaseConnection(settings) {
    ... // подготовительные шаги
    var connection = DBDriver.openConnection();
    if (connection == null) {
        throw DatabaseException;
    }
    return connection;
}

function readData() {
    var dbSettings = readSettings();
    try {
        var connection = openDatabaseConnection(dbSettings);
    } catch (var exception) {
        print "ERROR: could not open database";
        LOG.log(LogLevel.Fatal, exception);
        exit (-1);
    }
    ... // с этого момента мы знаем, что у нас есть работающее соединение
}
```

[1] Во многих низкоуровневых библиотеках и сейчас ради повышения производительности применяются коды ошибок.

Функция `openDatabaseConnection()` проверяет, запущен ли драйвер и установлено ли соединение с базой данных. Если значение `connection` равно нулю, то есть соединение не удалось, функция с помощью `throw` бросает исключение (типа DatabaseException). Функция `readData()`с помощью блока `try` дает знать, что она хочет ловить и обрабатывать исключения при соединении с базой данных. Если бы исключение бросила `readSettings()`, то оно бы не обрабатывалось `readData()`, так как обращение к `readSettings()` производится снаружи блока `try`. В нашем простом примере `read-Data()` лишь выдает пользователю сообщение об ошибке, заносит исключение в журнал и завершает работу программы, однако функция может также попросить пользователя проверить логин и пароль для соединения с БД или спросить, должна ли она попытаться еще раз установить соединение.

Как именно выбрасываются и ловятся исключения, варьируется от языка к языку. Так, бросание может называться `raise` или `throw`, поимка — `catch`, `rescue` или `except`:

- Ruby: бросить исключение — `raise`, поймать исключение — `begin ... rescue Exception`;
- Java: бросить исключение — `throw new Exception("Message")`, поймать исключение — `try{...} catch (Exception ex)`;
- JavaScript: бросить исключение — `throw "Message"`, поймать исключение — `catch (ex)`;
- PHP: бросить исключение — `throw new Exception("Message");`, поймать исключение — `catch(Exception $e)`;
- Python: бросить исключение — `raise Exception("Message")`, поймать исключение — `try ... except Exception`.

Правила обращения с исключениями также различны. Так, например, в Python принято открывать файл и делать в нем запись, не проверяя, было ли открытие файла успешным и доступен ли он для записи. Если у программы нет прав для записи или открытие не было успешным по другим причинам, то бросается и ловится исключение.

На другом полюсе находится Java, в сообществе пользователей которого исключения используются только для непредвиденных и серьезных ошибок. Обрывающееся соединение с базой данных стало бы поводом для исключения, но вместе с тем принято проверять, было ли открытие файла успешным, чтобы по возможности избегать исключений при записи.

Оба лагеря, конечно, убеждены, что их путь правилен, а другая сторона занимается ерундой.

Поэтому вникните в правила вашей целевой платформы, касающиеся обращения с исключениями, — часто там можно обнаружить интересные возможности. Например, в Android вы можете установить собственный «всепрограммный» обработчик исключений, к которому направляются все непойманные исключения. С помощью этого обработчика вы можете заносить исключения в журнал и предотвращать сбои вашего приложения.

В принципе, можно подавить исключения, но, пожалуйста, не делайте этого:

```
try {
    var connection = openDatabaseConnection(dbSettings);
    } catch (var exception) {
    }
    ... // и программа выполняется дальше
```

Хотя программа и обращается к пустому блоку `catch`, но поскольку он не прерывает выполнение программы, то оно продолжается, как если бы исключения не возникало. Этот способ лишить самого себя ценной диагностической информации во всех языках считается дурным тоном. Если вы поймали исключение и установили, что ничего не можете с ним сделать, то просто бросайте его дальше в блок `catch` с помощью `throw` — кто-то порадуется, если сможет его поймать. Если этот путь неприемлем, так как программа не должна прерываться и вы не уверены, будет ли исключение где-нибудь поймано, то хотя бы занесите его в журнал.

Основные правила

- Если вы имеете дело с ресурсами за пределами вашей программы, то есть с файлами, серверами, а также USB-периферией, то вам придется считаться с исключениями.
- Не подавляйте исключения, лучше дайте им привести к сбою программы.
- Пишите блоки try/catch только в том случае, если ваша программа может предпринять что-то осмысленное, когда исключение туда попадет.
- Получше разберитесь с особенностями используемого языка, касающимися работы с исключениями.

Обработка ошибок

Описанные ранее исключения — это область обработки ошибок, предусмотренная для серьезных ошибок и исключительных ситуаций, однако есть и менее серьезные ошибки, которые могут обрабатываться по-другому.

Например, пользователь может вызвать программу командной строки вроде `less`, но ввести в нее путь к файлу, которого не существует в файловой системе. В этом случае бросать исключение довольно бессмысленно, так как программа командной строки завершится с каким-нибудь непонятным для пользователя сообщением об ошибке вроде `uncaught Exception in Line 184`. Еще пример: функция в вашей программе ищет в базе адресов пользователя с определенной фамилией, но никого не находит. В этом случае также не стоит бросать сообщение `UserNotFoundException`, так как данная ошибка возникает при нормальной работе программы.

Итак, во время выполнения программы могут возникать ошибки разной степени тяжести:

- неверный ввод данных пользователем;
- поиск, который не приводит к нахождению результатов;
- пустые массивы или строки с длиной 0 передаются функциям, которые хотят осуществить операции с массивами или строками;
- в переменной содержится текст, хотя она должна содержать число;
- программа пытается разделить на 0;
- вместо объекта функции передается `NULL`;
- прочитанные файлы содержат данные, которые не соответствуют ожидаемому формату (например, XML в CSV-файле);
- файлы не могут читаться или записываться или же вообще не существуют;
- обрывается соединение с сервером или базой данных;
- компьютер вспыхивает и сгорает.

Даже самые тривиальные ошибки, такие как неверный ввод данных пользователем или пустые результаты поиска, стоит помещать в журнал. Познакомьтесь с библиотеками логирования (см. раздел «Чего не нужно делать самостоятельно» главы 19) и заносите в журнал ошибки, возникающие во время выполнения программы. Используйте разные уровни занесения в журнал: для самых тяжелых случаев — fatal, для тяжелых — error, а для менее серьезных ошибок — warn или info. Если возникает много ошибок сразу или программа ведет себя неожиданно, то эти журналы могут стать важным диагностическим средством. Прикрепляйте к сообщению журнала как можно больше информации, например, имена функций.

В зависимости от степени серьезности ошибки применяются следующие стратегии.

- Просто записать ошибку в журнал, немного подождать и еще раз попробовать выполнить операцию, так как это могло быть временное нарушение.
- Возвратить из функции код ошибки или `NULL` вместо значения.
- Бросить исключение.
- Завершить программу.

Ошибки ввода данных пользователем по возможности должны перехватываться еще пользовательским интерфейсом с выдачей пользователю содержательного сообщения об ошибке, чтобы он знал, что он ввел что-то не то, и знал где. Поиск, который не привел к нахождению результата, также не должен обрабатываться как ошибка. В этом случае программа должна работать дальше и просто ничего не делать.

Бессмысленные параметры, такие как пустые строки в функциях, работающих со строками, как правило, тоже не должны обрабатываться как ошибки. Обычно в таких случаях в зависимости от типа возвращаемого параметра возвращают или `0`/`NULL`, или пустую строку.

С неподходящими типами данных можно обращаться по-разному в зависимости от случая. Если вашей функции передается число вместо строки, то пусть она просто преобразует его в строку, а не выдает ошибку. Да, Ruby, мы грозно

поглядываем именно в твою сторону[1]. Если же, наоборот, арифметической функции передается строка, то она должна возвратить `NULL` или `NaN` (то есть Not a number — «результат не является числом») или бросить исключение. В этом отношении традиции разных языков сильно различаются. В то время как PHP, а также JavaScript, как известно, грешат тем, что в случае крайней необходимости конвертируют почти все, что состоит из битов, в `0`, `1` или `false`, языки с сильной типизацией, такие как C++ или Java, часто предотвращают подобные ошибки еще при компиляции с помощью проверки типов. Если такое все же произойдет во время выполнения программы, то лучше бросить исключение, чем проводить автоматическое преобразование типов. Ориентируйтесь в своих привычках на традиции языка.

Деление на 0 и другие подобные преступления против математики в разных языках также обрабатываются по-разному. C, будучи старым машиноориентированным языком, издает в таком случае так называемый сигнал SIGFPE — нечто вроде «всепрограммного» кода ошибки, обрабатываемого операционной системой. JavaScript считает, что 0 / 0 будет `NaN`, тогда как, например, 1 / 0 будет, по его мнению, `Infinity`, в чем он согласен с математически-статистическим языком R. Многие другие языки бросают исключение. Ваши стратегии обращения с делением на 0 должны соответствовать специфике языка: вы должны или проверять, является ли результат деления числом или чем-то вроде `NaN`, или производить вычисление в блоке `try/catch`. Проще всего, конечно, так фильтровать вводимые данные, чтобы случаев деления на 0 вообще не возникало. А если они все же непредвиденно возникнут, то это можно будет считать указанием на ошибку программирования.

Если функции в качестве параметра передается объект, то часто имеет смысл с самого начала проверить, не является ли этот объект `NULL`, и в этом случае сразу прервать выполнение программы и возвратить также `NULL`:

```
function formatForOutput(userRecord) {
    if (userRecord == null) {
        return null;
    }
    ... // иначе функция может работать нормально
}
```


В разделе «Утверждения» далее в этой главе мы опишем, как эту проверку можно реализовать по-другому, в форме утверждения, чтобы обращение, содержащее столь грубую ошибку, приводило к немедленной и явной ошибке выполнения программы.

Возвращайте `NULL` только в том случае, если у вас есть основания ожидать появления входного параметра `NULL` при нормальном выполнении программы, но если такого быть не должно, то напишите утверждение или бросьте исключение.

---

[1] Если написать в Ruby в целях отладки puts "Number of Records: "+numRecords, то в качестве результата вам будет выдано сообщение об ошибке, написанное мелким шрифтом: can't convert Fixnum into String. Язык не будет сам конвертировать число в строку и прикреплять его к первой строке.

Этот образ действий называют также принципом быстрого сбоя (fail fast): хорошо обнаруживать ошибки как можно раньше, даже если программа в результате дает сбой.

Если программа получает входные данные в неверном формате, то она не может записать их в файл, если происходит обращение по несуществующему пути файла, то в большинстве случаев правильным будет выдать пользователю сообщение об ошибке. Программы командной строки должны в таком случае сразу завершаться, тогда как программы с графическим интерфейсом должны продолжить работу, чтобы пользователь мог выбрать другой файл.

Если случаются действительно из ряда вон выходящие вещи, например, внезапно обрывается соединение с базой данных, то функция, которая хотела воспользоваться этим соединением, должна бросить исключение. Возможно, у программы есть шанс восстановить соединение, если она поймает такое исключение. Если это возможно, она должна это сделать. Если это невозможно, то программе следует предупредить пользователя, что она больше не может работать, и затем завершиться. Еще один пример, в котором исключение было бы правильным выбором: пользователь отсоединил внешний винчестер, на котором его программа открыла файл. В этом случае пользовательский интерфейс также должен поймать исключение и проинформировать пользователя.

Но особенно важно для мобильных приложений считаться с тем, что из-за плохого приема обращения к серверу могут быть неудачными. В таком случае им стоит не завершаться, а еще пару раз повторить запрос. Если все равно ничего не получится, они должны проинформировать пользователя.

Если вы показываете пользователю сообщение об ошибке, то оно должно быть оформлено как можно более человеколюбиво, а не так, как показано на рис. 26.1.

Рис. 26.1. Сообщение об ошибке, которое мало чем поможет

Признаемся, что это очень трудно, писать осмысленные сообщения о неизвестных ошибках. Но все же вам следует показывать пользователю более или менее подробное описание ошибки и возможных мер по ее устранению. Мы не знаем, что значит «ошибка 100», но было бы хорошо, если бы в сообщении было написано примерно следующее: «Сервер не ответил. Возможно, на нем неполадки, но причина может быть также в вашем интернет-соединении. Проверьте, можете ли вы открыть другие сайты. Если да, то повторите попытку через несколько минут или обратитесь в службу поддержки (указаны контактные данные)».

Основные правила

- Заносить ошибки в журнал всегда, когда это возможно, и делать это осмысленно.
- Ошибки ввода делают люди, и поэтому в ответ на них должны выдаваться сообщения об ошибке, которые может прочитать человек.
- Функции должны проверять свои входные параметры и реагировать на бессмысленное и неожиданное быстро и решительно: прервать выполнение программы или бросить исключение.

Состояние и отсутствие сохранения состояния

Протокол без сохранения состояния (stateless protocol) обрабатывает каждый запрос так, как будто отправляющий запрос клиент и отвечающий на него сервер никогда раньше не имели друг с другом дела. Internet Protocol (IP, межсетевой протокол) и Hypertext Transfer Protocol (HTTP, протокол передачи гипертекста) являются протоколами без сохранения состояния.

Для сравнения рассмотрим телефонные разговоры с сотрудниками кол-центра: когда вы изложили свою проблему, вам говорят: «Минутку, я вас соединю» — вы попадаете в руки нового сотрудника и приходится излагать ему проблему с самого начала. Состояние предыдущего разговора, как и возможные промежуточные результаты, следом не передаются. Дружелюбные по отношению к клиентам кол-центры иногда сохраняют состояние — это значит, что второй сотрудник приветствует вас словами: «Здравствуйте, госпожа Пассиг, я знаю, что у вас проблемы с роутером и что вы уже убедились, что штепсель вставлен в розетку». Иногда это состояние сохраняется даже в течение длительного срока: «В наших документах значится, что эта проблема уже возникала у вас четыре раза в прошлом году».

Преимущество *отсутствия сохранения состояния* в том, что задачи можно без проблем распределить между многими сотрудниками кол-центра или разными частями сервера, если это необходимо, причем выполнение задач реализуется проще и с меньшими затратами ресурсов: никто не должен что-то запоминать или сохранять, каждый запрос обрабатывается просто — раз-два и готово. Сервер, который просто выдает запрошенные данные без сохранения состояния, может справиться с большей нагрузкой, чем тот, который должен при каждом запросе вытаскивать из гор хлама документы, необходимые для работы именно с этим пользователем.

Так как HTTP является протоколом без сохранения состояния, но часто нужно идентифицировать возвращающихся пользователей или предоставлять определенным пользователям более широкие права, чем остальным, то существуют различные методы управления сессиями. К ним относятся, например, cookies, по которым сервер узнает, что тот или иной запрос исходит от одного совершенно определенного клиента. Тогда сервер может извлечь из своей базы данных касающиеся сессии

подробности, такие как имя пользователя и уровень прав, и воспользоваться этими данными.

В Web у вас нет выбора: HTTP — протокол без сохранения состояния, и если вам нужно сохранить состояние, то придется делать это с помощью cookies и управления сессиями.

ID, GUID, UUID

Люди всему хотят дать имя, это правило справедливо как для их собственных детей и домашних питомцев, так и для континентов и записей в базе данных. То, что в обыденной речи мы называем именем, в программировании называется ID (identifier, идентификатор) — это нечто, что позволяет отличать какую-либо единицу от других похожих единиц. В программировании ID применяются очень часто, например, в качестве индексов массивов, которые позволяют однозначно идентифицировать HTML-элементы на одной странице или находить записи из одной таблицы БД. У хороших ID есть пара особенностей: они стабильны в течение длительного времени, то есть не изменяются с изменением версии ПО, они всегда одного типа (то есть или строка, или целое число, но не смесь) и они настолько уникальны, что их нельзя перепутать.

Даже если уникальность нельзя повысить, все же существуют ID с уникальностью различной степени.

Есть локальные ID, которые могут быть использованы только один раз в рамках одного локального контекста (например, функции или массива). Для таких локальных ID часто используются просто порядковые номера. Например, записи в таблицах реляционных баз данных часто получают ID, которые однозначно идентифицируют записи только одной таблицы. В других таблицах этой же БД могут быть ID с такими же именами. Уникальность сохраняется, пока ID используются только для той таблицы, для которой должны. ID1 из таблицы 1 вполне может иметь то же значение, что и ID1 из таблицы 2, и путаницы не возникнет. Вне контекста таблицы значения ID1 и ID2 неуникальны, что, однако, вообще не проблема, так как база данных знает, из какой таблицы какие данные происходят.

```
insert into employee (id, firstname, lastname) values (1, "Ганс", "Майзер");
insert into department (id, name) values (1, "Развлечения");
select * from employee;
+----+-----------+----------+
| id | firstname  | lastname |
+----+-----------+----------+
| 1  | Ганс      | Майзер   |
+----+-----------+----------+
select * from department;
+----+--------------+
| id | name         |
+----+--------------+
| 1  | Развлечения   |
+----+--------------+
```

Как видите, и в таблице `employee`, и в таблице `department` есть записи с ID1. Вы также можете видеть, что это совсем не запутывает базу данных, так как в запросе указываются как таблица, из которой должны быть считаны данные, так и ID записей.

UUID (universally unique ID) — это ID, которые, напротив, обладают абсолютной уникальностью. Даже при обмене данными с другими программами или смешении данных из различных источников для своих записей можно не задумываясь применять UUID, и они не перепутаются с другими. Для UUID не может быть речи об использовании порядковых номеров, так как нет центрального пункта выдачи UUID, который следил бы за тем, чтобы каждый ID использовался лишь единожды. Поэтому применяются или большие случайные числа, или цифровые отпечатки (см. раздел «Хеш-коды, дайджесты, цифровые отпечатки» далее в этой главе), и вы полагаетесь на то, что статистические вероятности защитят вас от двойного использования ID. Обычно размер UUID составляет не менее 128 бит, а это такое огромное количество комбинаций, что по теории вероятности во всем мире ни один ID не будет сгенерирован больше одного раза. Существует международный стандарт, который регулирует генерацию и представление UUID, но в большинстве случаев эти детали неважны. Даже при разработке больших систем о UUID нужно знать, по сути, только то, где их можно получить, и то, что их невозможно перепутать. Если вам нужны надежные, однозначные ID, то стоит воспользоваться библиотекой UUID, которая их генерирует. В каждом языке есть как минимум одна такая библиотека.

GUID (globally unique ID) — это особая форма UUID, которые применяются для различных данных в Windows. GUID часто употребляется как синоним UUID, даже когда речь идет не о UUID, сгенерированных Windows.

Если вы используете в своих программах ID, то лучше не полагаться на чужие ID, а генерировать их самому, чтобы они были однозначны в вашем контексте, то есть в вашей программе, ее системе хранения и выдачи данных. Особенно при сохранении чужих данных в вашу БД вы снова и снова будете сталкиваться с вопросом о том, можно ли считать чужие ID однозначными, неизменными с течением времени и способными сохранять свой формат. Если нет абсолютной уверенности, то обязательно нужно дополнить чужой ID записи собственным ID. Например, может возникнуть идея, что комбинация имени и фамилии может послужить хорошим ID. Это неверно по двум причинам: во-первых, люди могут изменить свои имя и фамилию, во-вторых, может быть двое Гансов Майеров. Номера налогоплательщика также не являются хорошими ID, поскольку у пользователя может появиться новый номер. Номер паспорта — тоже плохая идея, так как потерянный паспорт заменяется новым паспортом с другим номером. О данных кредитных карт и счетов вообще не может быть речи, так как, во-первых, они меняются, а во-вторых, если кто-нибудь взломает вашу базу данных, он сможет пойти и наделать покупок на деньги ваших клиентов.

Для домашнего использования часто можно отказаться от применения UUID из специальной библиотеки для генерации однозначных ID. Если данные содержатся в массиве, то можно просто использовать индексы элементов массива, расположенные по возрастанию. Если вы записываете данные на диск, то часто можно

применять временную метку с миллисекундным разрешением или так же повышать переменную счетчика. А если вы используете базу данных, то можно просто оставить ей задачу генерации первичных ключей для каждой записи.

Языковые семьи

В зависимости от способа подсчета говорят о том, что существует от 500 до более чем 8000 языков программирования. Очень велика разница между крайне машиноориентированными языками, такими, как язык ассемблера, и языками, которые разрабатываются для совершенно определенных областей, таких как искусственный интеллект. Но все же можно создать классификацию языков программирования, распределив их по группам.

Существует также целое множество схем такой классификации, многие из которых неважны для практики, поэтому мы ограничимся беглым обзором.

Среди языков программирования часто выделяют две макросемьи.

- Императивные языки, в которых программист говорит компьютеру, как тот должен решать проблему.
- Декларативные языки, в которых программист говорит компьютеру, какую задачу он хочет выполнить.

На первый взгляд разница небольшая, однако эти подходы несопоставимы. Императивные языки (это распространенные языки, о которых сразу вспоминают, когда слышат слова «язык программирования», например PHP, C, Java или Python) устроены, как кулинарный рецепт: возьми то-то и сделай с этим то-то. К достижению желаемого результата ведут отдельные шаги, расположенные в хронологическом порядке в виде команд (отсюда и слово «императивный»). Например, чтобы удалить всех пользователей с именем Бернд, вам придется использовать цикл, который будет перебирать элементы массива `users` по одному.

```
for (var userIndex = 0; userIndex <= users.length; userIndex++) {
    var user = users[userIndex];
    if (user.firstName == "Bernd") {
        users.delete(userIndex);
    }
}
```

В декларативных языках (примерами которых являются SQL, регулярные выражения и в широком смысле также CSS) все происходит как при заказе в ресторане: мне, пожалуйста, пиццу номер 21 без лука, но добавьте побольше сыра. Хотя я знаю результат, мне не обязательно знать рецепт, и я не буду указывать повару в пиццерии, какие операции он должен совершить. Решающее различие: детали выполнения программы, а также последовательность операций в императивных языках определяет программист, а в декларативных подбирает сама программа.

Императивные языки позволяют программисту реализовывать любые тонкости и детали программы. В них большое значение имеют объекты, циклы и переменные — всегда, когда вы используете циклы `for` или `while`, вы программируете на императивном языке. В декларативных же языках вы задаете критерии фильтрации

в форме селекторов (см. раздел «Селекторы» далее в этой главе), и язык применяет эти фильтры к данным, например, в CSS — «все теги `<div>` класса `maincontent`», в SQL — «все записи в таблице `USER` с именем `Bernd`». Использует ли язык цикл, чтобы применить ваши фильтры к данным, или имеет в запасе еще какие-то трюки, вам неизвестно. В декларативных языках для изменения данных вы задаете операцию, которая выполняется над выбранными заранее данными, — в нашем примере на CSS это `background: white`, а в примере на SQL — `удалить`. Пример, приводившийся ранее, в SQL выглядел бы так:

```
delete from USER where FIRSTNAME = 'Bernd';
```

Никаких циклов, никаких переменных счетчика, а вместо них данные в таблице `USER`, критерий `where FIRSTNAME = 'Bernd'` и операция `delete` над выбранными записями.

Сейчас наиболее широко распространены императивные языки, и такое положение будет сохраняться еще долго. Каждый программист должен владеть хотя бы одним языком из этого большого семейства, потому что эти языки составляют основу программирования — от макроса в Excel до настольных программ.

Среди императивных языков различают процедурные, такие как C, и объектно-ориентированные, такие как C++, C# или Java. Чисто процедурные языки все больше теряют свое значение, они все чаще заменяются языками с неявной объектной ориентацией — языками, которые могут в значительной мере применяться в качестве процедурных языков, однако поддерживают и объектную ориентацию. Так как объектная ориентация, как описано в главе 23, является расширением процедурного подхода, о чисто объектно-ориентированных языках едва ли можно говорить: есть такие примеры, как Java, где процедурное программирование затруднено, однако другие языки, например PHP, Perl и JavaScript, более либеральны.

Но уже в качестве продвинутого новичка вам придется столкнуться с концепциями декларативных языков — не позже, чем когда у вас впервые возникнет необходимость сохранить ваши данные в БД, для чего придется иметь дело с SQL. А любому веб-разработчику не избежать знакомства с CSS и его селекторами.

Ранее считавшиеся эзотерическими и сильно завязанные на математике концепции функциональных языков также постепенно пришли в мейнстрим, поскольку JavaScript, к большому сожалению для многих поклонников чисто функционального программирования, — это крайне успешный гибрид функциональных и императивных языков. AJAX — технология JavaScript, которая позволяет догружать данные в веб-приложение, применяет принципы функционального программирования: функции, которая делает AJAX-запрос, передается функция обратного вызова, обращение к которой происходит только тогда, когда AJAX-запрос привел к появлению результата (или ошибки):

```
var successHandler = function() {
    if (xmlHttpReq.readyState == 4) {
        update_page_content(xmlHttpReq.responseText);
    }
}

var xmlHttpReq = new XMLHttpRequest();
```

```
xmlHttpReq.onreadystatechange = successHandler;
xmlHttpReq.send(getquerystring())
```

В строке `xmlHttpReq.onreadystatechange = successHandler` переменной `xmlHttpReq.onreadystatechange` ставится в соответствие не значение функции, как можно было бы предположить в императивном программировании, а сама функция — с тем чтобы объект `XMLHttpRequest` смог к ней позже обратиться. Тот факт, что функцию можно присваивать переменной в качестве значения, является решающим отличием функционального программирования.

Раньше существовало строгое разделение между скриптовыми и компилируемыми языками. Скриптовые языки интерпретируются — это означает, что языковой интерпретатор во время выполнения программы перемещается по исходному коду и выполняет каждую команду. Таким образом, на компьютере, на котором запускается программа, должен находиться исходный текст программы и должен быть установлен язык. В компилируемых языках программа задолго до своего выполнения превращается в работающую в ходе отдельного прогона компилятора, после чего она может запускаться и на других компьютерах (к этому типу относятся настольные программы вроде браузеров и текстовых редакторов). Скриптовые языки были медленными и использовались большей частью на серверах в качестве back-end; иногда понятие «скриптовый язык» использовалось как ругательство.

Со временем ландшафт стал менее однородным. С одной стороны, современные скриптовые языки тоже стали под шумок компилироваться, правда, во время выполнения программы. С другой стороны, многие классические компилируемые программы имеют возможности для скриптинга. Например, веб-браузеры — это уже не только C++, но и JavaScript, и веб-приложения заменяют компилируемые программы во многих областях.

Типы переменных

Есть языки программирования, в которых для каждого типа данных (чисел, букв, буквенных цепочек, указаний времени) предусмотрен собственный тип переменных. Таковы, например, C++, SQL и Java. Другие языки гораздо более небрежны и, с точки зрения программиста, сваливают все данные в один тип, который называется, к примеру, просто `var`. Примерами таких языков являются JavaScript и PHP. Хотя на самом деле эти языки обладают типами данных, тип переменной они изменяют динамически — тогда, когда это необходимо. По поводу того, какие языки лучше — те, которые требуют наличия типов данных в исходном коде (типобезопасные языки), или те, в которых не нужно задавать тип (языки со слабой типизацией), — долгие годы ведутся жаркие дебаты.

Преимущества типобезопасных языков в том, что уже компилятор может проанализировать типы. Это значит, что в выражении вроде `a = b` он сравнивает `a` и `b` и проверяет, относятся ли они к одному и тому же типу. Если нет, выдает сообщение об ошибке. Типобезопасные языки довольно надежно защищают разработчика от целого класса ошибок, допускаемых по рассеянности. Например, уже невозможно заниматься вычислением квадратного корня строки.

Четко оформить код помогают также объявленные типы переменных. `char` в C++ — это что-то совсем иное, чем `bool`, даже если они одной длины (1 байт). А если переменная относится к типу `float`, то понятно, что в ней можно ждать появления десятичной дроби, в то время как переменная `int` может содержать только целочисленные значения. В отличие от комментариев к коду это имплицитное оформление всегда актуально, так как является частью кода. Если вы измените код, чтобы расширить область значений переменной `int` до `long`, то в языке с типизацией это тут же станет заметно. Если вы написали комментарий вроде «здесь мы ожидаем только числа не более 2 млн», но потом это ожидание перестало быть актуальным, то вам необходимо изменить комментарий в соответствии с новым положением вещей, а этого очень часто не происходит.

Языки со слабой типизацией не требуют от разработчика объявлять тип переменных. Однако в большей части таких языков типы переменных все же определяются во время выполнения программы. Это важно, так как `a + b` во многих языках, например в PHP и JavaScript, означает, что должно быть выполнено сложение, если `a` и `b` — числа. Если одна из двух переменных — строка, то `b` добавляется к `a` и результат также является строкой.

Примеры:

```
var $v1 = 1;
var $v2 = 1;
print ($v1 + $v2);
```

Результат равен 2.

```
var $v1 = "1";
var $v2 = 1;
print ($v1 + $v2);
```

Результат равен 11.

Поведение гибких типов переменных во многом соответствует нашим интуитивным ожиданиям: так, например, числа можно без проблем записывать как текст, распечатывать или присоединять к другим строкам, но мы не рассчитываем на то, что строка `Ганс Майзер` превратится в число и с ней будут производиться вычисления. Тот факт, что разные типы данных имеют разную специфику, отражается в языках со слабой типизацией следующим образом: они отказываются от объявления типов переменных и в случае необходимости автоматически меняют один тип на другой. Если число можно без проблем рассматривать как строку, то должно быть возможно и сравнение переменной с числовым содержимым и строковой переменной. В типобезопасных же языках программист должен был бы сам превратить одну из них в переменную другого типа.

Типобезопасные языки намеренно делают различные типы переменных несовместимыми, чтобы разработчик уже при написании (или, самое позднее, при компиляции) программы обратил внимание на то, что пытается извлечь квадратный корень из пирога с творогом.

Неважно, предпочитаете ли вы жесткие рамки типобезопасных языков или дикую свободу языков со слабой типизацией, в любом случае программируйте в соответствии с концепцией языка. Правда, и в типобезопасном языке можно превратить все числа в строки и использовать только строковые переменные,

чтобы избежать пары призывов произвести преобразование численных типов в другие, а других — в численные. Но в результате получится очень плохой код, так как вы просто откажетесь от проверки типов, которая является частью концепции языка.

Если вам приходится в языке со слабой типизацией постоянно проверять с помощью цикла, можете ли вы считать элементы массива числами, это значит, что при разработке программы вы слишком мало думали. Даже в языках со слабой типизацией массивы не должны, словно чуланы, содержать все подряд — лучше создайте массивы с элементами одного типа в каждом или научитесь создавать массивы, содержащие объекты, и придайте каждому объекту различные типы переменных в качестве свойства.

Короче, не так:

```
var users = [];
users.push ("Ганс Майзер ");
users.push (1);
```

а лучше вот так:

```
var userIds = [];
var userNames = [];
userIds.push (1);
userNames.push ("Ганс Майзер ");
```

А еще лучше вот так:

```
var users = [];
var user = new UserObject(id: 1, name: "Ганс Майзер");
users.push (user);
```

Это уже само по себе весьма плохая идея — в императивном языке засунуть все данные в один родовой тип данных, такой как строковый. Но совсем плохой эта идея становится в SQL, самый обобщенный тип в котором называется BLOB. Это простой контейнер для двоичных данных почти любого размера. Так как все виды данных в компьютере в конченом счете двоичные, то в формате такого контейнера вы можете сохранять любые данные. Искушение для всего использовать BLOB поначалу велико, если вы как разработчик приложения рассматриваете базу данных просто как место сохранения для своих данных — будто обычный файл. Дело в том, что это позволяет экономить трудозатраты: вместо того чтобы разбираться в тонкостях различных строковых типов в разных диалектах SQL (TEXT, TINYTEXT, VARCHAR(255), VARCHAR2(4000) и т. д.), вы просто для всего создаете BLOB.

Например, вы как раз работаете над системой управления товарами и колеблетесь, стоит ли сохранять высоту и ширину товара как число с плавающей запятой без указания единиц измерения или все же лучше записывать их в поле базы данных вместе с пометками «см» и «мм». В этом случае идеальным решением кажется определить поля как BLOB. Даже если в ходе работы над проектом будет решено все же записывать единицы измерения в базу данных, в схеме базы данных уже ничего не придется менять.

Однако подвох обнаруживается тогда, когда вы хотите предоставить пользователям возможность осуществлять поиск товара и по таким характеристикам, как

размер. В идеальном случае размер был бы определен как целое и записывать вместе с ним единицы измерения было бы запрещено. Тогда поиск можно было бы осуществлять очень просто:

```
$stmt = $dbh->prepare("SELECT * FROM ITEMS WHERE width = ?");
$stmt->execute($width);
```

Если единицы измерения все же были записаны вместе с размерами и поэтому поле было определено как VARCHAR(100), то все по-прежнему еще не очень плохо. Код останется неизменным, только поиск будет длиться ощутимо дольше, так как на поиск по строкам в базе данных требуется больше времени, чем на поиск по целым числам.

Если же вы написали все в BLOB, то при поиске сперва нужно превратить каждую запись в строку:

```
$stmt = $dbh->prepare("SELECT * FROM ITEMS WHERE CAST( width AS CHAR ) =
?");
$stmt->execute($width);
```

В худшем случае это может парализовать механизм идентификации БД и время поиска по базе данных увеличится в огромное количество раз.

Разделение содержания и представления

Когда средства передачи информации еще полностью основывались на бумаге, передаваемое содержание, то есть мысли и информация, были привязаны к внешней форме. Нельзя было ни увеличить шрифт, ни выбрать другой тип шрифта, ни — что было бы очень приятно людям с нарушением зрения — понизить контрастность изображения. Радио и телевидение мало что в этом изменили.

Компьютер же способен хранить содержание отдельно от его внешнего представления. Всякий, кто изменяет шрифт документа в текстовом редакторе, уже испробовал эту особенность. На самом деле с этой точки зрения текстовые редакторы — это все же маленький шаг назад. В бронзовый век IT тексты сначала писались в таких системах, как TeX, и только потом компилировались в документ с определенным внешним видом. Можно было отдельно изменить исходник документа и скомпилировать новую версию.

Место этих систем верстки сегодня во многих областях занял Интернет, в котором документы больше не компилируются, а вместо этого снабжаются в браузере визуальными атрибутами, такими как цвет, тип шрифта, кегль и рамки. В HTML области также разделяются на части с помощью разметки: автор пишет команды прямо в тексте, но при выдаче пользователю они не отображаются, так как не предназначены для читателя-человека. В случае с HTML это теги, которые заключаются в угловые скобки <>[1].

---

[1] Такое разделение логического членения текста и визуальных команд не является обязательным. Были различные подходы, в том числе в ранних версиях Mac OS, в рамках которых визуальная информация хранилась в двоичных файлах за пределами текста. Но в дальнейшем эти подходы не получили распространения.

В разделении структуры документа и содержания, с одной стороны, и команд по поводу внешнего представления, с другой стороны, есть свои преимущества.

- Содержание можно легко сделать доступным для совершенно различных приборов — с появлением производительных браузеров на смартфонах с их маленькими дисплеями это стало особенно важно.
- Поисковым машинам и другим программам гораздо легче обрабатывать документы, так как их не интересует форма представления.

Принцип разделения функции и внешнего вида хорошо зарекомендовал себя не только в области работы с документами — он важен и для разработки ПО. Если вы хотите разработать программу, которая не только может переформатировать пару кусков данных, но должна работать в качестве веб-приложения или даже локально устанавливаемого приложения, то вам придется побеспокоиться об интерфейсе и о том, как связать его с back-end программы.

Хоть и нельзя просто прикрепить к программному коду таблицу стилей, которая будет генерировать интерфейс, но front-end и back-end можно разделить. Front-end генерирует интерфейс пользователя, а back-end (также называемый бизнес-логикой) — это часть, которая хранит и перерабатывает данные, записывает их в БД, обращается к URL или валидирует данные, вводимые пользователем.

Разделение на сервер разработки и «продакшн»-сервер

Часто веб-приложения пишутся в соответствии со следующей моделью.

- Сервер, на котором приложение будет в дальнейшем запущено, является также сервером разработки. Программист связывается с помощью FTP-программы с сервером и скачивает оттуда и загружает туда файлы программы (или текстовый редактор со встроенным FTP создает видимость редактирования информации прямо на сервере, в то время как на самом деле все по-прежнему скачивается и закачивается).
- Изменения вносятся локально, после сохранения измененный файл загружается на сервер и там тестируется.
- Если что-то идет не по плану, то снова локально производится изменение и повторяется загрузка на сервер.

Хотя эта модель и радует своей простотой, в ней есть тот недостаток, что время приема-передачи (round-trip time), которое требуется на сохранение, тестирование, исправление, новое сохранение, составляет несколько секунд. Слова «несколько секунд» звучат безобидно, но в первую очередь именно неопытным программистам часто требуется совершить множество попыток, прежде чем удастся добиться того, чтобы код делал примерно то, что должен. И вот несколько секунд уже превращаются в минуты, в течение которых можно было бы сделать что-то менее тупое, например вынести макулатуру. Нескольких секунд может хватить, чтобы вырвать человека из ментального состояния потока — состояния концентрации, в котором вы одновременно держите в голове представление о том, что хотите реализовать,

и ту точку, в которой сейчас находитесь. Если уж вы из него выпали, то может пройти довольно много времени, прежде чем вам снова удастся в него вернуться, и так несколько секунд могут перерасти в несколько четвертей часа.

Выходом могло бы стать следующее: отказаться от загрузки программы на удаленный сервер, вместо этого установить веб-сервер на рабочий компьютер и заниматься разработкой только локально. В наши дни любая операционная система позволяет установить локальный веб-сервер с поддержкой PHP, а если это нежелательно, то можно без проблем запустить в VirtualBox или похожей программе виртуальную машину, в которой есть веб-сервер, языки программирования на выбор и SFTP-доступ. Если вы работаете с такой конфигурацией, то время приема-передачи ощутимо уменьшается.

Кроме того, соединение с удаленным сервером может попросту оборваться. Тот, кто хоть раз пытался в поезде через UMTS устранить маленькую ошибочку, знает, как неприятно бывает, когда полоса загрузки просто зависает. Сразу возникают интересные вопросы вроде: «Приводит ли прерванная на половине загрузка к тому, что на сервере появится полфайла?» А когда что-то действительно идет не так, вас начинает преследовать мысль: «У меня же есть резервная копия... или нет?» По спине пробегают мурашки. И не зря, так как при таком методе разработки резервной копии, как правило, нет.

При написании веб-приложения с помощью локального сервера разработки и «продакшн»-сервера придется приложить несколько больше усилий при создании возможностей конфигурации, так как то, что разрабатывается локально, когда-нибудь должно перекочевать на «продакшн»-сервер, а там переменные среды и пути обычно все же другие. Это не обязательно недостаток, и можно превратить это в преимущество, так как в этом случае вы с самого начала можете позаботиться о сборе таких данных, специфических для серверов, в файлах конфигурации. Это в целом хорошая идея, и в будущем она может принести пользу, так как однажды вы, возможно, захотите поменять хостера.

Наряду с этим у разделения на сервер разработки и «продакшн»-сервер есть то бесценное преимущество, что можно заниматься разработкой более смело. Если вы встроили в приложение необъяснимые ошибки, то в случае необходимости можно все стереть и извлечь прежнюю версию из системы контроля версий. А если ее нет — взять резервную копию. Или в худшем случае версию с «продакшн»-сервера. И все это время приложение спокойно продолжает работать на «продакшн»-сервере, как урчащий котик.

Селекторы

Значительная часть разработки ПО состоит в том, чтобы разбить крупные массивы данных на более мелкие множества. Если вы (как программист, пишущий ПО для обработки изображений) имеете дело с неструктурированными данными (в данном примере — с кучей пикселов), то вам придется учитывать каждый отдельный блок данных, если понадобится, например, заменить все красные пикселы зелеными. Это длительный и некрасивый процесс (если только вы уже не стали программистом получше, который знает удобные трюки).

Чаще всего вы имеете дело с сильно структурированными данными, которые уже носят различные имена, имеют свои прически и рубашки, то есть обладают удобными для сравнения качествами, на основании которых их можно группировать. Если вы, например, хотите написать систему управления адресами, то можете исходить из того, что во всех записях будет содержаться одно и то же имя. Далее можно исходить из того, что поиск в массиве данных по имени будет одним из самых частых случаев использования БД. Структурированные базы данных как раз хорошо подходят для поисковых запросов вроде «дай мне всех Майеров из Берлина». Однако такой гибкий поиск совсем не просто запрограммировать, так как критерии и их количество могут колебаться. Если написать программу для поиска всех записей массива адресов, для которых верно `last_name = "Maier"` и `city = "Berlin"`, еще довольно просто, то все становится сложнее, если `last_name` должна включать также `Meier` и `Mayer`. (Maier, Meier и Mayer — разные способы написания одинаково звучащей немецкой фамилии Майер.) Затем приходит проверка и хочет из этого массива получить только тех, для которых верно `payment_status = "prepaid"`. Или тех, которые являются членами организации более года. Теперь дело уж точно принимает нехороший оборот.

Чтобы помочь этой беде, были созданы, например, языки запросов к базе данных, такие как SQL. С их помощью можно осуществлять довольно гибкий поиск сильно структурированных данных в больших объемах информации. Тайна успеха таких языков — так называемые селекторы.

Селектор представляет собой список, который описывает, какие элементы массива данных мы были бы рады получить для дальнейшей обработки, тогда задачей программы становится просеять данные с помощью селекторов.

Везде, где есть возможность отфильтровать данные с помощью селекторов, стоит это делать. Эта своего рода прикладная теория множеств делает программы значительно короче (а часто и быстрее) и очень существенно сокращает число ошибок.

Пример: если вы хотите представить пользователю форму, то ошибки при ее заполнении должны приводить к появлению сообщения об ошибке, которое будет с очевидностью восприниматься как таковое. Например, можно закрасить фон какой-либо области красным, чтобы четко указать на ошибку.

Вот соответствующий HTML-код:

```
<div>
    <div class="error">Ошибка!</div>
    <div class="ok">OK!</div>
</div>
```

Теги `<div>` с классом `"error"` должны быть закрашены красным.

Для нахождения введенных элементов в форме сначала нужно, не применяя язык селекторов, найти все теги `<div>` и перебрать их по одному, чтобы найти правильную область. В JavaScript это выглядело бы примерно так:

```
var nodelist = document.getElementsByTagName('div');
for (var i = 0; i < nodelist.length; i++) {
    var node = nodelist[i];
    var cName = node.className;
```

```
    if ((cName) && (cName.indexOf( 'error') != -1)) {
        node.setAttribute ('style', 'background: red');
    }
}
```

Чтобы сократить этот код, можно подключить библиотеку JavaScript под названием jQuery, которая позволяет применять селекторы[1]. Если вы подключили ее в `<head>`, то в JavaScript хватит следующей команды:

```
$('div.error').css('background', 'red');
```

Это работает так.

`$('div.error')` ищет в данном HTML-документе все теги `<div>` с классом `error`, то есть применяется селектор, который работает многоступенчато. Сначала он ищет все теги `<div>` — в этом примере их нашлось три. Потом сужает это множество до тех тегов, у которых в атрибуте `class` стоит `error`, — их два. И наконец, в результате этой операции фильтрации CSS-свойству `background` придается значение `red`.

Пример на SQL:

```
select street from users where town = 'Berlin';
```

Здесь фильтрация также происходит многоступенчато. Сначала база данных понимает, что мы интересуемся записями из таблицы `users`. Таким образом, записи в других таблицах нерелевантны для этого поиска. Затем база данных находит те записи, у которых в столбце `town` содержится строка `Berlin`. Для этих записей возвращается только значение столбца `street`.

Селекторы — это часть декларативного программирования. С их помощью вы задаете вид объектов, с которыми хотите работать, и вам не нужно искать их самому. Декларативное программирование может существенно укоротить код и помогает снизить вероятность появления в нем ошибок. В то же время такой код бывает трудно читать и отлаживать, потому что в сложные многоступенчатые селекторы нельзя вставить команду `print`, которая выдала бы результат на одной из промежуточных ступеней. При использовании селекторов результатом всегда является множество. Даже при однозначном поиске элемента, который предельно четко обозначен с помощью уникального ID, в качестве результата вы получите множество с одним элементом.

Пространства имен

Крупные программы обычно составляют из частей, написанных разными авторами, например из библиотек функций и/или классов и кода, написанного специально для проекта. Из-за этого возникает опасность, что двое программистов применят одно и то же имя для глобально видимой функции или переменной. Естественно, это может произойти и тогда, когда вы хотите снова использовать свой код, например строковые функции из проекта A и те удобные функции списков из проекта B.

---

[1] Если вам не нужно заботиться о старых версиях браузеров, то можно сделать это и без jQuery — с помощью getElementsByClassName().

И если вы понимаете, что в обоих модулях есть функция с именем `sort()`, которая делает в разных проектах несколько разные вещи, то у вас возникает проблема. В лучшем случае компилятор или интерпретатор тут же обращает на это ваше внимание. Эту проблему называют конфликтом имен (name clash), который заключается в столкновении двух функций или глобальных переменных с одним и тем же именем.

Традиционное решение заключается в добавлении к имени глобально видимой переменной приставки, написанной большими буквами. Эта приставка, которая часто состоит только из двух знаков, содержит отсылку к модулю или проекту, из которого она происходит. Таким образом, из `sort()` и `sort()` получаются `PROJECTA_sort()` и `PROJECTB_sort()` или, более компактно, `PA_sort()`и `PB_sort()`.

Общим решением являются пространства имен (namespaces), которые определяются эксплицитно. Большое преимущество этого решения в том, что по вашему коду не будут раскиданы уродливые большие буквы. Вместо этого в начале файла вы определяете, какие пространства имен хотите импортировать. Затем компилятор проверяет, действительно ли вы используете код из указанного пространства имен.

К примеру, система управления контентом может генерировать HTML-страницы как для статей, так и для постов в блоге, причем требования для разных типов статей различны. В Ruby в такой ситуации нужно было бы определить два модуля:

```
module Article
    class Page
    #...
    end
end

module Blog
    class Page
    #...
    end
end
```

Код сохраняется в `article.rb` и `blog.rb`. Тогда, чтобы создать новую страницу, нужно написать:

```
require "blog"
blogpost = Blog::Page::new
```

или

```
require "article"
article = Article::Page::new
```

Таким образом, страница однозначно определяется своей принадлежностью к модулям `Article` или `Blog`.

Когда не слишком опытные программисты, такие как авторы обеих функций `sort()` из первого примера, отказываются от использования пространств имен, это называется global namespace pollution — загрязнение глобального пространства имен. Давайте вместе сохраним чистоту глобального пространства имен! Оно у нас только одно, и от нас зависит, каким оно достанется нашим детям!

Объектно-ориентированные языки предлагают значительно более элегантное решение: так как в них каждый класс составляет собственное пространство имен, то есть функции и переменные экземпляров должны иметь однозначные имена только внутри класса, то здесь проблема конфликта имен возникает реже и касается только имен классов. Использование классов как пространств имен не решает проблему полностью, а скорее смягчает ее и отодвигает на второй план. Во многих объектно-ориентированных языках каждый класс включен еще и в иерархию пакетов (например, `java.util`), благодаря чему создается еще более обширное пространство имен.

Область видимости переменных

У переменных есть область действия, или область видимости, по-английски называемая scope. Только тогда, когда переменная находится в области видимости (in scope), ей можно присваивать значения, а также считывать ее значения. Вне области видимости язык программирования ее как бы не узнает (переменная невидима). Области видимости в большинстве языков обладают следующими свойствами.

- Они иерархичны. Есть области видимости высокого уровня (глобальные, то есть видимые по всей программе), и есть области видимости поменьше (локальные, то есть распространяемые только на одну функцию, но не за ее пределы).
- Переменные с глобальной областью видимости видимы в локальных областях видимости, но не наоборот.
- Локальные переменные имеют преимущество перед глобальными. Предположим, вы используете переменную `loop` для глобального цикла, но в функциях, к которым обращается этот цикл, у вас также есть локальные переменные с именем `loop`. Если теперь в одной из этих функций считать значение `loop`, то что будет возвращено — значение локального или глобального цикла? Ответ таков: преимущество имеет локальная переменная, то есть в функции более низкого уровня глобальная переменная невидима, так как затенена локальной. Если вы в функции присваиваете значение локальной переменной, то на глобальную переменную это не влияет. Но вне низкоуровневой функции программа видит глобальную переменную, а не локальную, так как имеет дело с областью видимости более высокого уровня.
- Переменные в рядом расположенных локальных областях видимости одного уровня иерархии не мешают друг другу. Если и в функции A, и в функции B используется переменная с именем `loop`, то в функции B видна только собственная переменная, но не переменная из функции B.

Хотя поначалу невозможность считывать значения любых переменных откуда угодно кажется обременительным ограничением, у областей видимости есть большие преимущества: видимым является только то, что действительно необходимо в данной функции, и вы не влезаете в дела других функций, случайно переписывая значения их переменных.

Использовать как можно меньше глобальных переменных — это в целом хорошая практика, так как иногда бывает ужасно трудно понять, кто именно и где именно меняет значения переменных. Чем больше вы работаете локально, тем легче удерживать проблемы в определенных границах. Функции — это локальные области кода, им передаются параметры, и они должны работать только с этими параметрами и своими локальными переменными. Если вы включаете в функции глобальные переменные, то ломаете эту замечательную структуру (см. также главу 14).

Следуя этой логике, создатели многих языков (например, Java) полностью убрали из них глобальную область видимости. Но если язык, на котором вы программируете, не таков (например, JavaScript, PHP или Perl), то все же по возможности не создавайте переменные в глобальной области видимости.

Утверждения

Утверждения (assertions) используются для защиты от ошибок из-за неверных входных данных. При этом значение, которое было получено в действительности, сравнивается с ожидаемым.

В функциях, которые рассчитывают получить в качестве входных параметров определенные значения, такую проверку следует выполнять в начале функции. Эти проверки называются проверками работоспособности (sanity-checks). Следующий цикл должен выдавать первые *n* клиентов из массива, желаемое число записано в `numPersons`:

```
function getFirstNCustomers (allCustomers, numPersons) {
    var foundCustomers = new Array();
    int loop = 0;
    while(foundCustomers.length < numPersons) {
        var customer = allCustomers[loop];
        if (customer != null) {
            foundCustomers.add (customer);
        }
        loop = loop + 1;
    }
    return foundCustomers;
}
```

Все работает хорошо, пока в `allCustomers` как минимум столько клиентов, сколько требует `numPersons`. Таким образом, если мы определяем массив клиентов:

```
var customers = new Array();
customers.add(new Customer("name" => "Ганс Майзер", "id" => 1));
customers.add(new Customer("name" => "Бернардетта Айзен", "id" => 2));
customers.add(new Customer("name" => "Тиночка фон Лурк", "id" => 3));
```

и затем обращаемся к функции таким образом, чтобы она возвратила две первые записи из этого массива, то получим:

```
>getFirstNCustomers (customers, 2)
>Customer("name" => "Ганс Майзер", "id" => 1)
>Customer("name" => "Бернардетта Айзен", "id" => 2)
```

Проблема этой функции станет очевидной, если мы при тех же входных данных захотим получить четырех клиентов вместо двух. В этом случае цикл `while` (во многих языках) превратится в бесконечный, так как `loop` когда-нибудь превысит длину массива `customers`. Каждая следующая считанная в строке `var customer = allCustomers[loop];` запись клиента в таком случае будет иметь значение `null`, в массив `foundCustomers` не смогут добавляться другие записи и условие выхода из цикла никогда не выполнится.

Обычно в таких случаях выдается лишь столько клиентов, сколько их имеется, но бывают случаи, когда столь негибкое поведение желательно, так как слишком большое значение в `numPersons` является сигналом о том, что в других частях программы произошла ошибка. В языке C, например, подобное наивное решение было бы уязвимостью, позволяющей считать содержимое памяти, к которому у программы, возможно, не должно быть доступа. Во избежание намеренных эксплойтов (то есть использования проблем с безопасностью) или ошибок программирования в нашем случае должны всегда выполняться два допущения:

- требуемое количество записей не может быть больше длины входного массива `allCustomers`;
- входной массив не может содержать записей со значением `null`.

Если представить эти параметры в форме, читаемой машиной, то программа сможет во время выполнения указать на нарушение заданных условий. Для проверки этих так называемых инвариантов (то есть неизменных предварительных условий) во многих языках программирования есть `assert()`. Утверждения снабжают функцию когтями и зубами, которыми она защищается от ошибок и манипуляций. Точное обозначение и синтаксис в разных языках разные, но принцип всегда похож:

```
assert (allCustomers.length >= numPersons, "Error: requested number exceeds
Array length" );
```

`assert()` чаще всего принимает два аргумента: сначала условие, которое должно быть проверено, а потом сообщение об ошибке, которое выдается в случае нарушения условия.

Поэтому надежная версия функции выглядела бы так:

```
function getFirstNCustomers (allCustomers, numPersons) {
    assert (allCustomers.length >= numPersons);
    var foundCustomers = new Array();
    int loop = 0;
    while(foundCustomers.length < numPersons) {
        var customer = allCustomers[loop];
        assert (customer != null)
        foundCustomers.add (customer);
        loop = loop + 1;
    }
    return foundCustomers;
}
```

Функция стала очень требовательной и прерывает программу с помощью исключения, если вы хотите считать большее количество записей клиентов, чем их вообще имеется, или если запись в массиве `customers` имеет значение `null`.

Хотя записывать инварианты функций в комментариях к коду и полезно, но комментарии не могут быть автоматически проверены во время выполнения программы и могут устареть. Если предварительные условия вашей функции изменились, а вы используете утверждения, то нарушение утверждения, которое произойдет в результате, заставит вас снова думать о функции и ее инвариантах.

Всегда используйте утверждения:

- когда вы уверены, что определенный случай никогда не произойдет. Если он все же произойдет, то вы будете оперативно проинформированы о логической ошибке;
- когда ошибка столь значительна, что продолжение работы программы не имеет смысла;
- когда вы работаете с данными, которые должны подчиняться определенной схеме. Строго проверяйте, соответствуют ли данные правилам, это защитит вас от искажения данных. Особенно в тех ситуациях, когда вам нужно обрабатывать большие массивы данных, у вас едва ли будет вероятность вручную найти в них ошибку;
- когда вы чувствуете искушение написать в комментарии: «...не должно происходить никогда». Когда-нибудь это произойдет, комментарий к коду не сможет этому помешать;
- в начале функции, чтобы принудительно определить область значений, которые могут принимать переменные;
- в конце функции, чтобы проверить, что результат имеет смысл, то есть относится к определенной области значений.

Стоит добавить, что утверждения — это мощное оружие. Если оно бьет мимо цели, то программа завершается, а это должно происходить, только когда действительно необходимо. Например, если ваша программа хочет записать результаты вычисления в файл, но этот файл защищен от записи и поэтому попытка записи оказывается неудачной, то действительно имеет смысл завершить программу. Иначе данные не запишутся, а пользователь ничего об этом не узнает. В этом случае `assert(file.canWrite())` — хорошее решение.

Если же утверждения использует ваш интерфейс, чтобы предотвратить ввод пользователем некорректных данных, это менее целесообразно. Если он сделал опечатку в форме ввода данных или забыл что-то указать, то программа должна любезно обратить на это его внимание, а не выдать мерзкое сообщение об ошибке и умереть.

Утверждения не являются необходимыми в том случае, когда вы знаете, что ошибка все равно вызовет исключение или прерывание программы, например, если вы хотите открыть соединение с базой данных, однако сделать это не удается. Хотя вы и можете проверить состоянии с помощью `assert(dbConnection.isValid())`, но если соединение не смогло открыться, то это приведет к исключению либо прямо при попытке открыть соединение (тогда программа вообще не дойдет до вашей проверки с помощью утверждения), либо, самое позднее, при первой операции с БД, которую программа попытается осуществить через отсутствующее соединение.

Кроме того, утверждения играют большую роль в главе 16: от них зависит успешность модульного тестирования.

Транзакции и откаты

Транзакции и откаты (rollbacks) призваны защитить от искажения данных в случае изменения нескольких записей.

Если в какой-либо записи в базе данных клиентов вы хотите изменить как имя, так и адрес, то легко можете сделать это с помощью единственного оператора `UPDATE`. Такая команда атомарна, она либо целиком выполняется, либо целиком не выполняется. Вам не нужно беспокоиться о том, что, возможно, было изменено только имя.

Если же, напротив, задействованы несколько команд, которые должны выполняться одна за другой, но связаны по смыслу, то без транзакций это может закончиться тем, что будут выполнены лишь первые шаги, а остальные — нет. Пример: нужно перевести деньги со счета A на счет B. Если в тот момент, когда деньги уже сняты со счета A, но еще не пришли на счет B, стажер споткнется о провод и вырубит электричество, то деньги пропадут. Поэтому такие многоступенчатые процессы защищаются с помощью транзакций.

Внешне транзакция ведет себя атомарно: либо она полностью выполняется, либо происходит откат. Это значит, что все произведенные изменения аннулируются и процесс полностью возвращается к исходной точке. Так, в нашем примере деньги в случае неудачной транзакции вернулись бы на счет A. В отличие от обычных SQL-команд в транзакции должны быть объявлены начало и конец, чтобы у базы данных была возможность разом откатить все связанные между собой команды.

Вне баз данных транзакции можно встретить сегодня в основном в файловых системах: современные файловые системы довольно хорошо защищены от ситуаций, когда, например, внезапно отключается свет. Даже если программа прямо в этот момент производит запись в файл, файловая система все равно не будет повреждена, потому что цепочка «создать, открыть, записать, закрыть файл» протоколируется в транзакции, в так называемом журнале. Если отключилось электричество, то потом операционная система видит, что операция с файлом не была завершена, и удаляет фрагмент файла из каталога. Конечно, записанные данные теряются, но вам уже не нужно, как раньше, исправлять содержимое винчестера или что-то устанавливать заново.

Идея транзакций, заключающаяся в том, что нужно иметь решение, позволяющее минимизировать ущерб от неполадок, может быть полезна также в деле разработки ПО. Например, если ваша программа скачивает и затем обрабатывает данные с сервера, то может быть целесообразным сначала полностью загрузить файлы, а потом начинать обработку. Если скачивание прервется, то вы сможете удалить данные, успевшие загрузиться, и начать все заново. Если же вы уже частично обработали данные, то могут возникнуть трудности с тем, чтобы бесследно удалить их.

Хеш-коды, дайджесты, цифровые отпечатки

Хеш-коды, или цифровые отпечатки, представляют собой алгоритмы, способные перегнать практически сколь угодно большие файлы в почти однозначно определяющее их число фиксированной длины. Конечно, с математической точки

зрения невозможно присвоить всему — от одной буквы до всей информации целого процессингового центра — неповторимые значения, но это и не является задачей алгоритмов хеширования. Их задача — лишь генерировать цифровые отпечатки, которые могут считаться уникальными для целей практического использования.

Одной из областей их применения программистами являются быстрые сравнения очень больших цифровых файлов. Например, у вас уже есть тысячи или даже миллионы картинок и вы получаете еще одну и хотите знать, не находится ли уже эта картинка в вашей коллекции. Конечно, вы можете просто сравнить новую картинку со всеми имеющимися, но сравнение со всеми файлами займет довольно много времени. Если же вы вычислите для картинки хеш-сумму, то вам нужно будет лишь выделить для нее всего 20 байт на каждую картинку, например, в базе данных. Если вам поступает новая картинка, то вы вычисляете ее хеш-сумму и сравниваете с уже сохраненными. Если два хеш-кода равны, то с вероятностью, граничащей с уверенностью, одинаковы и картинки. Поэтому хеш-суммы называют также цифровыми отпечатками пальцев.

Распространенные алгоритмы хеширования подчиняются простым правилам.

- Одинаковые входные файлы, пропущенные через определенный алгоритм хеширования, дадут одинаковые цифровые отпечатки.
- Разные входные файлы с чрезвычайно высокой вероятностью дают разные цифровые отпечатки. Могут происходить так называемые хеш-коллизии — ситуации, когда два разных входных файла дали в результате одинаковый хеш-код, но длина хеш-кода и алгоритмы подобраны так, что с человеческой точки зрения это практически невозможно.
- Длина цифрового отпечатка хотя и различается от алгоритма к алгоритму, но для определенного алгоритма всегда одинакова — неважно, насколько большим был входной файл.
- Это не алгоритмы сжатия, поэтому по цифровому отпечатку нельзя восстановить исходный файл. Можно легко вычислить хеш-сумму для определенного файла, но практически невозможно определить исходный файл для какой-либо хеш-суммы.
- Небольшое изменение входного файла приведет к появлению совершенно другого цифрового отпечатка, большое изменение — тоже. Поэтому нельзя считать подобие хеш-сумм мерой подобия входных файлов.

В криптографии хеш-коды служат электронным подтверждением подлинности: для текста высчитывается хеш-код, который затем шифруется и сохраняется. Если в оригинальном тексте произойдут какие-то изменения, то зашифрованный хеш-код уже не подойдет к нему. Благодаря этому получатель может проверить, не производились ли манипуляции с текстом, вычислив его цифровой отпечаток и сравнив с хеш-кодом оригинала. Шифрование хеш-подписи гарантирует, что она не будет изменена. Подобные тексты с цифровой подписью можно передавать через Интернет или сохранить в архиве и даже по прошествии многих лет иметь возможность обнаружить искажение. Конечно, можно было

бы зашифровать и весь текст, защитив его тем самым от подделок, но в случае с большими текстами это не слишком практично, так как в этом случае их размер удвоился бы.

Вот еще некоторые области применения хеш-кодов.

- В качестве контрольной суммы, чтобы распознавать ошибки при передаче данных.
- В качестве уникального ID определенной версии файла. Системы контроля версий Mercurial и git используют алгоритм хеширования SHA-1 для различения версий файла. В биоинформатике генетические секвенции также иногда идентифицируются с помощью их хеш-суммы.
- Генерация случайных чисел. При этом начинают с любого числа и высчитывают для него хеш-код, потом — хеш-код этого хеш-кода и т. д. Хеш-функции при небольшом отклонении дают совершенно иные числа, поэтому распределение чисел в такой цепочке вычислений случайно.
- Сохранение паролей. Вместо пароля сохраняется только его хеш-код. Если пользователь хочет выполнить вход, то высчитывается хеш-сумма введенного им пароля и сравнивается с сохраненным хеш-кодом. Если хеш-коды равны, то пароль был введен правильно. Однако тут вам стоит учитывать замечания, сделанные в главе 25.
- Цифровые криптовалюты, такие как bitcoin, во многом полагаются на то, что вычисление хеш-кода происходит очень быстро, однако практически невозможно подобрать подходящее входное значение к определенному хеш-коду. Они используют хеш-коды для защиты счетов и транзакций.

Как разработчику, хеш-алгоритмы могут пригодиться вам во всех случаях, когда содержимое файла или довольно объемный отрывок текста являются ключевой информацией, — для того чтобы различать их. В программах, обслуживающих блоги, или в системах управления изображениями вы можете хешировать содержимое и использовать получившиеся цифровые отпечатки как уникальные ID — в качестве имен файлов или в базе данных, неважно. В отличие от временных меток хеш-коды стабильны в течение долгого времени: если вы упорядочиваете файлы по дате их изменения, то вам неизвестно на 100 %, что произойдет с этими датами изменений, если вы переедете на другой сервер или придется вернуться к резервной копии.

Хеш-коды полезны также для унификации кусков информации разной длины. Если вы когда-нибудь захотите написать поисковую программу-робот, которая сохраняет содержимое страниц и использует URL страницы в качестве уникального ID, то быстро заметите, что URL могут оказаться неудобно длинными — фактически их максимальная длина не задана ни одним стандартом. Если же вы хешируете URL с помощью SHA-1, то будете точно знать, что после этого в вашем распоряжении окажутся уникальные последовательности длиной 20 байт. Вам все равно нужно будет где-то хранить нехешированные URL, однако число фиксированной длины может оказаться приятнее в качестве уникального ID.

CRUD и REST

Программы, которые производят манипуляции с базами данных, называют также CRUD-программами — по четырем основополагающим операциям, которые можно производить над содержимым баз данных:

- Create (создание);
- Read (чтение);
- Update (обновление);
- Delete (удаление).

Конечно, со считанными данными можно делать много других вещей, например, сравнивать их с другими данными и куда-нибудь загружать, но все это происходит уже вне рамок БД. Эти четыре основные операции не являются результатом произвольного выбора — это именно то, что можно делать с данными.

Так как речь идет об основополагающих операциях, то вы обнаружите их и вне баз данных, только под другими названиями. Все файлы на вашем компьютере подчиняются такому же основному принципу: вы можете их создавать и открывать, считывать из них информацию, сохранять информацию внутри них и когда-нибудь их удалить.

Пусть выделение этих операций при первом (да, признаться, и при десятом) прочтении не кажется сногсшибательным достижением мысли, однако иногда при разработке ПО оказывается полезно иметь их в виду. Если вы в своей программе перемещаете туда-сюда временные файлы, то это не что иное, как обновление метаинформации, а именно пути к файлам. Если вы открываете временный файл, считываете содержимое и записываете его в другой файл, то вы совершаете операцию «чтение» с временным файлом, операцию «создание» — с файлом назначения и, наконец, операцию «обновление» — также с файлом назначения в тот момент, когда записываете в него содержимое.

Этот принцип действует и в больших распределенных системах, таких как, например, Web: в системах управления контентом вы можете создавать новые записи, редактировать их и при необходимости удалять. Ну и само собой разумеется, что в Web вы можете читать HTML-страницы.

Разработчики протокола передачи гипертекста HTTP, лежащего в основе Web, реализовали эти четыре операции в данном протоколе, потому что они столь фундаментальны. Грубо говоря, HTTP работает так: клиент (например, ваш веб-браузер) посылает серверу запрос, состоящий из объекта назначения и глагола. Объект назначения определяется URL-адресом, а глагол — это один из определенных в HTTP глаголов: `POST`, `PUT`, `PATCH`, `GET`, `DELETE`.

Схему CRUD можно «спроецировать» на HTTP-глаголы примерно таким образом:

- Create (создание) — `POST`;
- Read (чтение) — `GET`;
- Update (обновление) — `PUT` и `PATCH`;
- Delete (удаление) — `DELETE`.

В 1990-е и 2000-е годы над этими HTTP-глаголами издевались как могли. Развитие Web ускорялось такими темпами, что посредственные поделки становились широко распространенным ПО. При этом разницу в значениях HTTP-глаголов часто оставляли без внимания: большие фреймворки, такие как сервлеты Java, использовали `POST` там, где правильнее было бы использовать `GET`.

В 2010-е годы произошло частичное возвращение к истокам Web. Это было связано с REST-движением (REST — Representational State Transfer, передача состояния представления), которое среди прочего подчеркивало, что для запроса данных с веб-серверов и манипуляций с ними должны применяться HTTP-глаголы, а URL-адресами не следует злоупотреблять, используя их для обозначения как самого действия, так и его объекта.

27 Что дальше?

> Судьба программиста: к концу дня не продвинуться ни на шаг, зато хотя бы узнать много нового.
>
> *Анна Шюслер/@quarkkrokettchen, Twitter, 3 мая 2011 года*

Вы прочитали эту книгу частично или даже полностью и все еще чувствуете себя плохим программистом. Возможно, вам стало даже яснее, чем в начале чтения, что на самом деле вы программируете не очень-то хорошо. Положение кажется безвыходным: пройдет еще 180 лет, прежде чем вы достигнете хотя бы уровня посредственности, и вы злитесь на себя самого и на авторов. Эта злость — неплохой знак. В «Системантике» Джона Голла есть такой рассказ об обучении дельфинов.

> Незадолго до того, как дельфин наконец понимает, в чем заключается суть нового трюка, у него портится настроение, он начинает плавать кругами, и в какой-то момент у него лопается терпение. Он выпрыгивает из воды и целиком обрызгивает дрессировщика. У психологов для описания этого явления существует понятие «когнитивный диссонанс» — чувство раздражения, возникающее из-за того, что что-то не так. Карен Прайор, дрессировщица дельфинов, чьи методы применяются по всему миру, называет эту реакцию learning tantrum — гнев обучения.

Учиться новому — это длительный процесс, и вполне возможно, что у вас возникает чувство, будто в вашем случае он длится еще дольше, чем обычно. Но продвижение вперед со скоростью улитки все же лучше, чем вообще никакого, а плохой код — это закономерный и забавный (многие говорят: единственный) путь к хорошему коду. Кроме того, ваш нынешний плохой код гарантирует, что через 20 лет вы сможете развлекать целые вечеринки[1] и рассказывать читателям тех сайтов, которые к тому времени займут место нынешних Coding Horror und Reddit, о грехах своей юности.

Быть может, вас утешит тот факт, что, очевидно, от 90 до почти 100 % людей, претендующих на должности штатных программистов, тоже не умеют программировать. Не просто не слишком хорошо программируют, а вообще не умеют.

---

[1] Ну ладно, по крайней мере, очень унылые вечеринки.

По крайней мере, об этом сообщает Джефф Этвуд, администратор Coding Horror, вот здесь: www.codinghorror.com/blog/2007/02/why-cant-programmers-program.html и вот здесь: www.codinghorror.com/blog/2010/02/thenonprogramming- programmer.html. Если вы знаете, как на самом деле обстоят дела с вашим программистским мастерством, то уже имеете некоторое преимущество перед этими соискателями. И если вы не избегнете полностью когнитивного диссонанса и время от времени будете испытывать гнев обучения, то, возможно, в один прекрасный день из вас по недоразумению выйдет путный программист, а может быть, даже хороший.

Кто такой хороший программист

Программиста делает хорошим не то, что он программирует на функциональных языках или владеет вычурными языками программирования, и не то, что он умеет особенно остроумно унижать неопытных коллег, задающих глупые вопросы. Все это не является ни причинами, ни следствиями компетентности — это лишь побочные эффекты, и то, обладает программист этими качествами или нет, зависит от его личных предпочтений. Мы ведь не говорим, что хороший водитель — тот, кто ездит быстрее всех или имеет машину определенной марки.

Хороший программист в состоянии построить мысленную модель своей программы. Он умеет читать код и может представить, что произойдет, если его выполнить. Он может в уме раскладывать сложные задачи на логические единицы, которые затем можно успешно воплотить в виде программных модулей. Хороший программист знает общие принципы программирования, которые может использовать и в незнакомых условиях, однако он не настолько прикипел душой к любимому языку, чтобы переносить из него удачные решения на все другие проблемы.

Хорошие программисты знают, что они пишут код в первую очередь для других программистов и лишь во вторую — для машины. Поэтому они в состоянии писать гуманный код — понятный, читаемый, снабженный комментариями в нужных и важных местах, обладающий настолько логичной структурой, что она раскрывается и перед другими. Хорошие программисты — это те, кто смог так хорошо понять собственный код, что теперь может объяснить его коллегам, даже если коллеги программируют хуже. Тот, кому нужно объяснить свой код менее опытному программисту, должен передать ему также множество фоновых знаний, объяснить различные концепции и заполнить пробелы. Для этого нужно действительно хорошо понять эти концепции самому.

Хорошие программисты работают скорее по нисходящей схеме. Они знают, какие задачи должна выполнять программа, и продумывают необходимую для этого архитектуру. Затем они реализуют верхние уровни программы, а более глубокие слои оставляют ненаписанными, с тем чтобы позаботиться о них позже. Хорошие программисты сначала пишут много кода, не тестируя его, потому что у них есть логичный план программы в голове. Тестирование для них заключается в том, чтобы находить реальные ошибки, в то время как плохим программистам оно нужно для того, чтобы посмотреть, работают ли их идеи. Иные программисты имеют в го-

лове настолько четкий план, что сначала задумывают тесты и только потом пишут программу.

Хорошие программисты тратят довольно много времени на то, чтобы следить за развитием концепций разработки ПО и языков программирования. Это позволяет им в дальнейшем активно применять эти концепции в своих проектах. Плохой программист на вопрос, почему он выбрал то или иное решение, вынужден отвечать: «Я сделал таким образом, потому что это показалось мне единственным выходом». Хороший же программист может сказать: «Я сделал таким образом, потому что это дает такие-то и такие-то преимущества и в дальнейшем принесет мне меньше неприятностей, чем другие варианты». (Есть и третий, вероятно, наиболее частый вариант: плохой программист выбрал свое решение по ошибочным причинам, но потом задним числом оправдывает их совершенно другими, притянутыми за уши аргументами.) Эта способность обосновывать свой выбор имеет много общего со способностью называть вещи своими именами. Если врач не знает названия органов и болезней, то с ним трудно о чем-либо вести разговор.

Хорошие программисты знают, что, несмотря на свою квалификацию, они не всегда правы. Они не участвуют в драках за территорию и при необходимости следуют чужим принципам программирования, даже если те не отвечают их предпочтениям. Для них ясно, что в деле разработки ПО играют роль не одни только рациональные аргументы, но и личные симпатии, антипатии и множество случайных факторов. Хорошие программисты считаются с этой данностью и деликатно обходятся с чувствами других участников проекта.

Если вы встретите такое просветленное существо, будьте с ним любезны, ищите его близости и время от времени угощайте его напитками. Тогда процесс вашего превращения в хорошего программиста займет не 180 лет, а, возможно, немного меньше. Но мы, конечно же, надеемся, что эта книга поспособствует тому, что вы продолжите продвижение по кривой обучения и ничто не сможет помешать вам писать код так, как вы можете. И при этом не забывайте, что за более или менее обозримый срок, в течение которого вы будете применять «лучшие практики», можно стать вполне сносным разработчиком.

Что еще можно почитать

1. *Макконнелл С.* Совершенный код. Мастер-класс. 2-е изд. — М.: Русская редакция, 2010. — 896 с. (*McConnell S.* Code Complete II. — Microsoft Press, 2004. Немецкое издание Microsoft Press Deutschland, 2005.) Если вы прочитаете только одну книгу, то пускай это будет она. Эта книга предназначена для продвинутых программистов, но по большей части будет понятна и новичкам.
2. *Босуэлл Д., Фаучер Т.* Читаемый код, или Программирование как искусство. — 2012. (*Boswell D., Foucher T.* The Art of Readable Code. — O'Reilly, 2011.) Если вы хотите избавить свои глаза от созерцания безобразных иллюстраций, то здесь найдете хорошее краткое изложение наиболее важных советов с простыми примерами.

3. *Гудлиф П.* Ремесло программиста. Практика написания хорошего кода. — М.: Символ-Плюс, 2009. — 824 с. (*Goodliffe P.* Code Craft: The Practice of Writing Excellent Code. — No Starch Press, 2006.)
4. *Фаулер М.* Рефакторинг. Улучшение существующего кода. — М.: Символ-Плюс, 2008. — 568 с. (*Fowler M.* Refactoring: Improving the Design of Existing Code. — Addison Wesley Professional, 1999.) Эта книга поможет понять, что даже специалисты делают все так же, как и обычные люди (и при этом действуют тоже очень по-разному).
5. *Голл Д.* Системантика: Библия систем (*Gall J.* Systemantics: The Systems Bible). Встречается также под названием «Библия систем: Пособие для начинающих по системам большим и маленьким» (The Systems Bible: The Beginner's Guide to Systems Large and Small). Незаменимый и очень веселый справочник об обращении со сложными системами. Быстрый путь к пониманию сути маленьких проектов по разработке ПО и больших компаний.

Благодарности

Работа над этой книгой затянулась более чем на пять лет, поэтому мы в первую очередь благодарим сотрудников издательства O'Reilly за большое терпение, особенно работавших с нами редакторов Кристину Хайте и Инкен Киупель, а также Еву Лакас — за терпеливую поддержку. Некоторыми страницами текста, а также многими исправлениями мы обязаны Яну Бёльше, который позволил прочитать себе вслух наполовину законченную книгу и все время терпеливо говорил: «Но ведь все это совершенно неправильно». Все остальные помощники перечислены далее в алфавитном порядке (за исключением тех, кого мы забыли, — мы всех благодарим как за их вклад, так и за снисходительность).

Dirk Ahlers (@dirkahlers), Matthias Bauer (@moeffju), @chinhzilla, Tobias Fiebiger (@scholt), Frollein (@dorfpunk), @gedankentraeger, Helge Grimhardt, Lukas Hartmann (@mntmn), Christian Heller (@plomlompom), Arne Janning, @ironmadna, Markus Kempken (@slowtiger), Mario Konschake (@Infinite_Monkey), Markus Krajewski, Fiona Krakenbürger (@Fotografiona), Roland Krause (@spitshine), Sonja Krause-Harder (@skh), Nils Dagsson Moskopp, Philipp Oelwein, Georg Passig, Rin Räuber (@rinpaku), Matthias Rampke (@matthiasr), Felix Rauch, @roarrrbert_we, Aleks Scholz (@dalcashdvinsky), Kai Schreiber, Anne Schüßler (@Quarkkrokettchen), Stefan Schwarzer, André Spiegel (@drmirror), Michael Spitzer, Cornelia Travnicek (@frautravnicek), Jan Varwig (@agento), Tim Weber (@scy), Lars Weisbrod (@6percentrecall), Ivo Wessel, Florian Felix Weyh (@FFWeyh).

Очень небольшие части этой книги были написаны в городе Клагенфурте, в квартире для бесплатного проживания лауреатов литературной премии «Штадтшрайбер» (квартира была ненадолго предоставлена Корнелией Травничек), и в рамках проекта «Штадтшрайбер» Института им. Гете в Сан-Паулу.

Указатель