97

Collective Wisdom
from the Experts

97 Things Every Programmer Should Know

O'REILLY®

Edited by Kevlin Henney

Содержание

97 вещей, которые должен знать каждый программист

Жемчужины мудрости для программистов, собранные ведущими специалистами-практиками.

Это перевод проекта '97 вещей, которые должен знать каждый программист'.

Если вы обнаружили ошибки или у вас есть какие-либо предложения, вы можете создать issue или сделать pull request в репозиторий.

Переведено пользователем ЖЖ под ником avl. Оригинал перевода можно найти здесь.

Автоматизируйте стандарт кодирования

(В оригинале - Automate Your Coding Standard)

Наверняка с вами это тоже случалось. В самом начале проекта у всех присутствует множество позитивных намерений. Часто эти намерения даже фиксируются в документах. Вещи, касающиеся кода, формируют стандарт кодирования. В ходе kick-off митинга технический лидер проходит стандарт кодирования по пунктам, и в лучшем случае каждый соглашается следовать этому стандарту. Однако по ходу проекта намерения потихоньку исчезают одно за другим. И когда проект завершаетсся, код выглядит совершенно беспорядочно, и при этом никто не может понять, как же оно так получилось.

Когда же что-то пошло не так? Вероятно, еще на kick-off митинге. Кто-то из команды просто не обратил внимания. Кто-то неправильно понял. А кто-то вообще не согласился и уже тогда решил писать по-своему. А те, кто изначально согласились придерживаться стандарта кодирования, но когда стали поджимать сроки, им пришлось чем-то жертвовать. А хорошо отформатированный код не приносит дополнительных баллов в глазах заказчика, который хочет дополнительную функциональность. И к тому же, следовать стандарту кодирования может быть скучно, если это не автоматизировано. Попробуйте как-нибудь расставить индентацию в беспорядочно написанном файле.

Но раз все так сложно, то зачем мы так хотим, чтобы все придерживались определенного стандарта? Этим мы можем хотеть избежать использования антипаттернов с целью предотвратить известные ошибки. К тому же, общий для всех стандарт делает работу на проекте проще, поддерживая высокую скорость разработки в течении всего времени. Из этого следует, что согласиться с принятым стандартом должны все – если один программист будет использовать три пробела для индентации, а другой четыре, то ничего хорошего не получится.

Существует множество инструментов для генерации отчетов о качестве кода и для документировании стандарта кодирования, но это еще не все. Автоматизировать нужно все, что касается стандарта кодирования и поддается автоматизации. Вот несколько примеров.

- Сделайте форматирование частью процесса сборки, чтобы оно выполнялось при каждой компиляции.
- Используйте статические анализаторы кода для обнаружения антипаттернов. Если какой-нибудь антипаттерн будет обнаружен, завершайте сборку, словно

произошла ошибка.
- Научитесь настраивать анализаторы для поиска ваших собственных антипаттернов, специфичных для вашего проекта.
- Не только измеряйте процент покрытия кода тестами, но и контролируйте результат. При слишком низком покрытии тоже аварийно завершайте сборку.

Старайтесь проделать это для всех важных частей. Автоматизировать все вам скорее всего не удастся. Для тех вещей, автоматизировать которые не получилось, составьте рекомендации в дополнение к стандарту, соответсвтие которому контролируется автоматически, разрешив не следовать безусловно этим дополнительным рекомандациям.

И наконец, стандарт кодирования должен быть гибким. По мере развития проекта меняются и требования, и то, что было логично в начале, может быть не столь разумным спустя несколько месяцев.

Автор оригинала - Filip van Laenen

Будьте предусмотрительны

(В оригинале Act with Prudence)

> *«Чем бы вы не занимались, делайте это с предусмотрительностью и думайте о результатах»* - автор неизвестен.

Каким бы легким не выглядело расписание в начале итерации, вам не избежать того, что иногда придется работать под давлением. Если вам приходится выбирать между «сделать хорошо» и «сделать быстро», то часто выбор падает на «сделать быстро» сейчас, а потом доделать как надо. Когда вы даете такое обещание себе, команде и заказчику, вы действительно в это верите. Но потом следующая итерация приносит свои проблемы, и вам приходится решать их. Для такой отложенной работы появился неологизм «технический долг» (technical debt). Говоря еще точнее, Martin Fowler называет его «преднамеренный технический долг», в противоположность «непреднамеренному».

У технического долга много общего с долгом обычным. Вначале вы получаете кратковременную выгоду, но потом вам приходится платить проценты до полной выплаты долга. «Срезание углов» в коде затрудняет дальнейшую модификацию или рефакторинг. Такой код становится рассадником ошибок и ненадежных тестовых сценариев. Чем дольше вы его оставляете без изменений, тем хуже он становится. К тому времени как вы соберетесь наконец-то все исправить, окажется, что уже из-за той самой неисправленной ошибки на проекте несколько раз было принято не самое лучшее решение, и из-за этих новых надстроек исправление не так просто сделать. И часто единственное, что реально вынудит вас таки вернуться и сделать исправление – положение из серии «хуже уже некуда». При этом исправление возможно станет столь сложным, что потребует слишком много времени или окажется слишком рискованным.

Иногда действительно приходится «повесить» на себя технический долг для того, чтобы успеть к дедлайну или реализовать требуемую функциональность. Старайтесь не попадать в такую ситуацию, но если вдруг избежать ее не получится, тогда просто сделайте это. Но! Обязательно отслеживайте технических долг и старайтесь оплатить его быстро, иначе проблемы начнут расти как снежный ком. Как только вы пошли на компромисс, внесите исправление этого в систему трекинга задач, чтобы не забыть о нем.

Если вы запланируете исправление долга в следующей итерации, потери будут минимальными. Оставление долга неоплаченным вызовет накопление процентов, и эти проценты тоже должны отслеживаться, чтобы визуализировать затраты. Это

подчеркнет эффект технического долга на проекте и позволит правильно расставить приоритеты. То, как именно подсчитывать и отслеживать «проценты» по техническому долгу, зависит от конкретного проекта, но делать этот учет вы обязаны.

Оплачивайте технический долг как можно быстрее. Действовать по-другому было бы неосмотрительно.

Автор оригинала - Seb Rose

В линкере нет никакой магии

(В оригинале - The Linker Is not a Magical Program)

Удручающе часто (в последний раз – буквально перед написанием этой статьи) программисты представляют процесс преобразования исходного кода в исполняемый файл как-то вот так:

1. Написать (или исправить) исходный код
2. Скомпилировать исходный код в объектные файлы
3. Происходит какое-то чудо
4. Появляется исполняемый файл

На шаге 3, конечно же, происходит линковка. Что же меня в этом так раздражает? Я работал в техподдержке десятилетиями, и постоянно получаю вопросы вроде:

1. Линкер сообщает что переменная определена более одного раза
2. Линкер сообщает что переменная является неразрешимым символом
3. Почему исполняемый файл такой большой.

И все это в перемешку со словами «кажется», «каким-то образом» и с ощущением непередаваемой сложности в голосе. Словно процесс линковки – это что-то столь магическое, что понять его могут только маги и чародеи. Процесс компиляции не вызывает столько вопросов – похоже, что программисты более осведомлены о том, как работают компиляторы, или по крайней мере знают, для чего они нужны.

Линкер – на удивление простая и незатейливая программа. Все, что она делает – это соединяет вместе секции кода и данных из объектных файлов, связывает символьные ссылки с их определениями, выбрасывает неиспользуемые символы из библиотеки и записывает исполняемый файл. И все. Никакой магии. Самая большая и сложная часть – это декодирование и генерирование ужасно переусложненных форматов файлов, что, однако, не меняет самой сущности линкера.

Пусть, например, линкер сообщает, что переменная определена более одного раза. Многие языки имеют как декларации, так и определения. Декларации обычно помещаются в заголовочные файлы:

```
extern int iii;
```

Таким образом создаются внешние ссылки на символ iii. Определение же создает само хранилище для этого символа, появляясь обычно непосредственно в коде, и выглядит примерно вот так:

```
int iii = 3;
```

И таких определений может быть только одно. А что, если определение появится в более чем одном файле?

```
// File a.c
int iii = 3;
```

```
// File b.c
double iii(int x) { return 3.7; }
```

Линкер сообщит о том, что символ iii определен более одного раза.

Если же iii встречается только в виде деклараций, то линкер сообщит о том, что iii является неразрешенной ссылкой.

Чтобы определить, почему размер исполняемого файла такой, каким получился, нужно взглянуть на файл .map, обычно генерируемый линкером. Этот файл – ничего более, чем список всех символов в исполняемом файле вместе с их адресами в памяти. Это покажет вам, какие модули были добавлены, и размеры каждого модуля. Теперь вы знаете, что «раздуло» ваш исполняемый файл. Часто вы не будете представлять, какая часть и где использует тот или иной «непонятный» модуль. Чтобы выяснить это, просто удалите подозрительный модуль из библиотеки и еще раз запустите сборку. Неразрешенные символы покажут, кто ссылается на этот модуль.

И хотя не всегда сообщения линкера столь просты и очевидны, в них нет ничего магического. Механизм всего этого очень простой, вам всего лишь нужно уточнить детали в каждом конкретном случае.

Автор оригинала - Walter Bright

Ваш заказчик говорит не то, что думает

(В оригинале - Your Customers Do not Mean What They Say)

Я еще ни разу не встречал заказчика, который не был бы рад описать мне в деталях то, что бы он хотел получить. Проблема только в том, что заказчики никогда не говорят всей правды. Обычно они не лгут, но при этом говорят как заказчики, не как разработчики. Они используют свою терминологию и свою предметную область. Они знают о значимых деталях. Они ведут себя так, словно вы уже 20 лет работаете в их компании. И это все накладывается на тот факт, что многие заказчики сами не знают точно, чего же они хотят. Некоторые хорошо представляют себе общую картину, но при этом не могут эффективно обсуждать детали. Другие не так четко представляют себе картину целиком, но при этом точно знают, чего именно они не хотят. В результате — как же вы можете создать продукт для кого-то, кто не говорит вам об этом всей правды? Все просто. Нужно лишь больше взаимодействовать с заказчиками.

Возражайте своим заказчикам, и делайте это почаще. Не нужно лишь повторять за ними то, что они говорят вам. Помните: они говорят не то, что думают. Я часто меняю слова в беседах с ними и смотрю на реакцию. Вы удивитесь, как часто под словом «Заказчик» подразумевается слово «Клиент». Человек, рассказывающий вам, что именно он хочет видеть в проекте, будет точно также менять термины «на лету» и при этом ожидать от вас, что вы уследите за тем, что он вам рассказывает. Это легко может сбить вас с толку, в результате чего пострадает ваш проект.

Обсудите темы с заказчиком несколько раз перед тем как вы решите, что уже все поняли, что он от вас хочет. Пробуйте переформулировать проблемы два-три раза. Обсудите то, что будет предшествовать и будет следовать за теми темами, которые вы обсуждали, чтобы лучше понять контекст. Если это возможно, пусть про одни и те же темы расскажут разные люди в разных разговорах. Практически всегда они расскажут вам разные истории, что откроет для вас новые отдельные, но при этом связанные друг с другом факты. Часто двое людей, говорящих про одно и то же, будут противоречить друг другу. Ваш лучший шанс на успех — это выяснить эти отличия до того, как вы начнете создавать свое суперсложное ПО.

Используйте визуальные подсказки во время обсуждений. Это может быть как доска с маркерами во время совещаний или же простая визуальная модель, так и сложный функциональный прототип. Общеизвестно, что использование визуальных материалов во время разговора помогает дольше удерживать внимание и повышает процент запоминаемой информации. Используйте это для достижения своего успеха.

В прошлом я работал мультимедиа-программистом в команде, делавшей отличные проекты. Наш клиент в подробных деталях описал свои мысли по поводу того, как должен выглядеть проект. Основная цветовая схема, обсужденная на нескольких совещаниях, содержала черный цвет фона. Мы решили, что все поняли. Команда дизайнеров начала создавать сотни графических файлов со слоями. Куча времени была потрачена на вылизывание конечного продукта. Поразительной силы откровение случилось в тот день, когда мы показали результат работы клиенту. Увидев его, первые слова, которые она сказала, были «Когда я говорила черный, я имела в виду белый». Как видите, ничего не может быть яснее, чем черное и белое.

Автор оригинала — Nate Jackson

Взаимодействие между процессами влияет на время отклика

(В оригинале - Inter-Process Communication Affects Application Response Time)

Время реакции критично для удобства использования. Ничто так не раздражает, как ожидание реакции ПО на ваши действия, особенно когда вам приходится это делать периодически. Мы чувствуем, что ПО безвозвратно тратит наше время. Однако, в современных приложениях редко когда обращают внимание на причины неадекватного времени реакции. Большинство литературы на тему оптимизации производительности все еще фокусируется на структурах данных и алгоритмах – вещах, которые конечно же могут иногда существенно влиять на производительность, но чаще всего влияют на нее весьма слабо. Особенно в современных многоуровневых промышленных приложениях.

Когда в таких приложениях возникают проблемы производительности, мой опыт говорит, что оптимизация структур данных и алгоритмов – это не то место, где надо искать улучшения. Время отклика гораздо сильнее зависит от количества взаимодействий между процессами. Безусловно, в системе могут быть и другие узкие места, однако чаще всего причина именно в удаленных взаимодействиях между процессами. Каждое такое взаимодействие вносит свой вклад в задержку, и эти задержки накапливаются, особенно если идут последовательно друг за другом.

Характерный пример – *ripple loading* в приложениях, строящих граф отношений между объектами. Ripple loading – это последовательное выполнение большого количества запросов к базе данных для выборки данных, необходимых для построения такого графа (см. Lazy Load в книге *«Паттерны архитектуры корпоративных приложений»* Мартина Фаулера). Если клиент базы данных – это сервер приложений промежуточного уровня, строящий веб-страницу, то запросы к базе данных выполняются последовательно в одном потоке. Их задержки суммируются, повышая общее время отклика. Даже если один запрос к базе данных занимает всего лишь 10 миллисекунд, для страницы, требующей 10 тысяч запросов (что не так уж и редко), задержка отклика составит уже 10 секунд. Другие примеры включают HTTP-запросы от веб-браузера, вызов распределенного объекта, сообщения типа «запрос-ответ» и другие подобные взаимодействия. Чем больше взаимодействий между процессами требуется для получения ответа на воздействие, тем большим будет время задержки.

Существует несколько относительно стандартных и хорошо известных стратегий для снижения количества взаимодействий между процессами в пересчете на получение одного ответа. Одна из стратегий – это Бритва Оккама, оптимизация интерфейса между процессами таким образом, чтобы во время взаимодействия передавались только те данные, которые требуются для достижения цели с минимальным количеством взаимодействий. Другая стратегия – распараллелить коммуникацию между процессами там, где это только возможно, при этом время реакции будет равно времени самого долгого взаимодействия. Третья стратегия – это кеширование результатов предыдущего взаимодействия, чтобы в будущем использовать данные из кеша вместо взаимодействия между процессами.

Когда вы проектируете приложение, учитывайте количество требуемых взаимодействий между процессами в ответ на каждое воздействие. Анализируя приложения с проблемами производительности, я много раз встречал соотношение порядка тысяч взаимодействий на один запрос. Снижая эту величину, неважно как именно – распараллеливанием или кешированием или еще какой-нибудь техникой, можно получить гораздо более существенный прирост производительности, чем оптимизируя структуры данных или алгоритмы сортировки.

Автор оригинала - Randy Stafford

Выучить язык - значит понять его культуру

(В оригинале - Don't Just Learn the Language, Understand its Culture)

В университете мне пришлось изучать иностранный язык. Тогда я думал, что английского мне будет вполне достаточно, и поэтому предпочитал регулярно спать на занятиях французского все три года. Спустя несколько лет я поехал в отпуск в Тунис. Официальный язык там – арабский, а из-за французского колониального прошлого широко распространен также и французский. На английском говорят лишь в туристических местах. Из-за незнания языка я проводил много времени у бассейна, читая «Поминки по Финнегану» Джеймса Джойса, мастерскую игру слова и формы. Его игра со смешением более чем сорока языков оказалась удивительной, хотя и изнуряющей вещью. Понимание того, как переплетения иностранных слов и фраз давала автору новые возможности для самовыражения – то, что я вынес и применяю в своей программистской карьере.

В своей книге *«The Pragmatic Programmer»* Энди Хант и Дейв Томас убеждают нас изучать новый язык программирования каждый год. Я пытался следовать их совету и спустя несколько лет получил опыт программирования на нескольких языках. Самым главным уроком этой мультиязычности оказалось то, что для того чтобы выучить язык, недостаточно выучить его синтаксис. Нужно понять его культуру. Писать на Фортране можно на любом языке. А чтобы на самом деле изучить язык, нужно в него вникнуть. Не извиняйтесь, если ваш C# код выглядит как одна большая функция main() с несколькими статическими helper-методами, а разберитесь, зачем нужны классы. Не избегайте лямбда-выражений в функциональных языках, если они кажутся вам слишком непонятными, а найдите силы разобраться в них.

Изучив сущности нового языка, вы удивительным образом обнаружите, что стали использовать уже известный вам язык немного по-другому. Я научился эффективно использовать делегирование в C# из программирования на Ruby, освоение всего потенциала .NET generics дало мне идею, как лучше использовать Java generics, а LINQ открыл мне второе дыхание для изучения Scala.

Вы также станете лучше понимать паттерны проектирования, используя разные языки. Программисты на C находят, что C# и Java сделали полезным шаблон iterator. В Ruby и других динамических языках вы по-прежнему можете использовать шаблон visitor, но ваша реализация уже не будет выглядеть как пример из книги Банды Четырех.

Кое-кто может возразить, что *«Поминки по Финнегану»* невозможно читать в принципе, в то время как другие будут приводить это произведение как пример элегантности стиля. С целью сделать книгу более доступной для чтения существуют ее переводы на единственный язык. По иронии судьбы, первым таким переводом стал перевод на французский. Код в чем-то похож на все это. Если вы напишете код на Wakes (так называют язык, на котором написана «Поминки по Финнегану») с одновременным использованием Python, Java и Erlang, то ваш проект будет полным хаосом. Если вы вместо этого изучите языки, чтобы расширить свои горизонты и получить идеи, как можно делать разные вещи разными способами, ваш код, написанный на вашем старом добром и любимом языке программирования будет только хорошеть с каждым новым освоенным языком программирования.

Автор оригинала - Anders Norås

Дайте проекту голос

(В оригинале - Let Your Project Speak for Itself)

Возможно, ваш проект использует систему контроля версий. Возможно, что у вас даже есть сервер интеграции, на котором постоянно работают автоматические тесты. Если это так, то это супер.

Вы можете добавить утилиты статического анализа кода в ваш процесс интеграции и собирать метрики. Метрики предоставляют обратную связь о различных аспектах вашего кода и его эволюции во времени. Как только вы начнете использовать метрики, вы определите границу, за которую не будете хотеть зайти. Например, вы решите не снижать процент покрытия тестами ниже 15%. Непрерывная интеграция позволит вам отслеживать подобные цифры, но вам все еще придется регулярно их проверять. А теперь представьте, что вы можете это делегировать непосредственно проекту, так, чтобы сам проект сообщал о нарушении установленных лимитов.

Вам нужно дать вашему проекту голос. Это может быть е-мейл или мессенджер, информирующий разработчиков о последнем результате измерений или о нарушении лимитов. Но еще более эффективным решением будет использование экстремальной обратной связи (ЭОС).

Идея ЭОС – в управлении каким-либо реальным физическим устройством: лампой, фонтанчиком, игрушечным роботом или даже запускателем ракеты, основанном на результате автоматического анализа. Как только ваши лимиты будут нарушены, устройство сработает. Например, загорится лампочка. Не заметить подобное срабатывание будет очень сложно.

В зависимости от типа ЭОС вы можете услышать сигнал о том, что сборка повреждена, увидеть «красный свет» или даже почувствовать запах плохого кода. Устройства могут дублироваться в различных локациях, если у вас распределенная команда. Вы можете, например, разместить светофор на столе менеджера проекта, который будет показывать текущий статус проекта – вашему ПМ это должно понравиться.

Позвольте вашей креативности выбрать подходящее устройство. Если вы еще не полностью забыли детство, используйте радиоуправляемые игрушки. Если хотите выглядеть более солидно, найдите настольную лампу соответствующего дизайна. Поищите варианты в Интернете. Все, что включается в розетку или имеет дистанционное управление, может использоваться в качестве ЭОС.

ЭОС работает как голосовой аппарат вашего проекта. С ним проект начинает взаимодействовать с разработчиками, одобряя или критикуя их согласно выбранным правилам. Вы даже можете использовать синтезатор речи и пару динамиков, и проект на самом деле обретет свой голос.

Автор оригинала - Daniel Lindner

Две ошибки могут взаимокомпенсировать друг друга (и найти это - очень тяжело)

(В оригинале - Two Wrongs Can Make a Right (and Are Difficult to Fix))

Код никогда не лжет, но он может сам себе противоречить. Иногда такие противоречия заставляют воскликнуть «Как это вообще может работать?».

В своем интервью один из дизайнеров ПО лунного модуля Аполлон 11, Алан Кламп (Allan Klumpp) рассказал, что в ПО контроля двигателей была ошибка, из-за которой во время посадки могла возникнуть нестабильность. Однако другая ошибка скомпенсировала первую, в результате чего ПО было использовано при посадке на Луну на кораблях Аполлон 11 и 12 до того, как ошибки были обнаружены и исправлены.

Представьте себе функцию, возвращающую успешность выполнения. Представьте, что она возвращает false в ситуации, когда должна вернуть true. А теперь представьте, что вызывающая функция не проверяет возвращаемое значение. В результате все будет работать до тех пор, пока кто-нибудь не заметит отсутствие проверки и не добавит ее.

Или представьте приложение, сохраняющее свое состояние в документе XML. Представьте, что один из узлов ошибочно записан как `TimeToLive` вместо `TimeToDie`. И вновь все будет работать до тех пор, пока код записи состояния и код чтения состояния будут содержать эту ошибку. Но исправьте ее в одном месте (или добавьте еще одно приложение, читающее этот документ уже правильно), и вот уже нарушена и симметрия ошибок, и работоспособность.

Если же две ошибки в коде проявляются одинаковым образом, то даже методология поиска неисправностей может дать серьезный сбой. Разработчик получает отчет об ошибке, находит первое проблемное место в коде, исправляет его и повторяет тест. Ошибка по-прежнему проявляется, ведь второе проблемное место все еще в коде! Он отменяет коррекцию, продолжает поиски и находит второе проблемное место, исправляет его – и вновь никакого эффекта (ведь первый дефект был возвращен назад). И разработчик продолжает поиски дальше, однако поскольку он уже обнаружил оба дефекта, но ошибочно решил, что причина не в них, поиски третьего дефекта, в реальности не существующего, могут оказаться очень долгими…

Такое случается не только в коде. Проблема может быть и в документации постановки задачи. И даже более – ошибка в коде может компенсировать ошибку в спецификации.

Аналогичный сценарий может затронуть и людей. Пользователи запоминают, что когда приложение просит нажать левую кнопку, для корректной работы нужно нажать правую. Люди запоминают это и даже передают это дальше: «Запомните, левая кнопка на самом деле означает правую!». Исправьте эту ошибку – и множеству людей придется переучиваться.

Единичные ошибки легко обнаруживаются и легко устраняются. Серьезная проблема возникает в случае множественных ошибок, требующих для исправления более одной коррекции. Отчасти так происходит и потому, что простые проблемы обычно исправляются сразу, а сложные накапливаются «до лучших времен».

Нет единой универсальной рекомендации, что делать в случае множественных проблем. Осведомленность о возможности такой ситуации, ясный ум и готовность учитывать все возможности помогут вам найти решение.

Автор оригинала - Allan Kelly

Делайте невидимое видимым

(В оригинале - Make the Invisible More Visible)

Многие аспекты невидимости очень точно отражают принципы программирования. В терминологии разработки полно соответствующих метафор, например, «механизм прозрачности», «сокрытие информации». Как ПО, так и процесс разработки может быть практически невидимым:

- Исходный код не имеет физической сущности, поведения и не подчиняется законам физики. Он становится видимым, будучи загруженным в текстовый редактор, но стоит редактор закрыть, как он вновь исчезнет. Если задуматься об этом достаточно долго, то вы начнете сомневаться в том, существует ли он на самом деле, как падающее дерево, которое никто не слышит.
- Запущенное приложение существует и имеет свое поведение, но оно практически никак не основано на исходном коде, его создавшем. Главная страница Гугла очень минималистична, главное же происходит «за кулисами».
- Если вы завершили разработку на 90% и при этом застряли на отладке последних 10%, то на самом деле вы вовсе не сделали 90%, ведь так? Исправление ошибок – это не движение вперед. Вам не платят за отладку. Отладка – это потери времени. Поэтому хорошо бы сделать эти потери более заметными, чтобы вы видели, куда тратятся ресурсы и планировали, как этого избежать.
- Если ваш проект все время шел по расписанию, а через одну неделю вдруг оказалось, что вы отстаете на 6 месяцев, у вас проблемы. И проблема не в том, что вы опаздываете на 6 месяцев, а в том, причина этой задержки была для вас невидимой! Если вы не видите продвижения, то скорее всего вы и не продвигаетесь.

Невидимость может быть опасной. Вы гораздо продуктивнее мыслите, когда у вас перед глазами что-то конкретное. Вы гораздо лучше управляете вещами и событиями, когда вы их видите и когда вы видите то, как они меняются.

- Написание юнит-тестов дает понять, насколько легко код модуля поддается модульному тестированию. Оно помогает обнаружить наличие (или отсутствие) тех или иных критериев качества, таких как минимальное количество связей и сильная связность.
- Выполнение юнит-тестов дает представление о поведении кода. Оно помогает обнаружить наличие (или отсутствие) качеств времени выполнения, таких как надежность и корректность.
- Использование системы назначения задач делает прогресс видимым. Задачи

могут быть видны как «неначатые», «в работе» и «завершенные» без связи со скрытым инструментом управления проектом и без принуждения программистов писать фиктивные отчеты.
- Инкрементальная разработка повышает видимость прогресса (или его отсутствия) повышая частоту обратной связи. Завершение готового к выпуску ПО показывает реальность, оценки – нет.

Лучше всего вести разработку при наличии множества постоянно видимых эффектов. Видимость вызывает уверенность в том, что вы на самом деле движетесь вперед, и делаете это так, как было запланировано, а не по воле случая.

Автор оригинала - Jon Jagger

Делайте ревью кода

(В оригинале - Code Reviews)

Вы должны делать ревью кода. Почему? Потому что это повышает качество кода и снижает процент ошибок. Но не совсем по тем причинам, о которых вы могли бы подумать.

Многие программисты не любят ревью кода, потому что имели ранее негативный опыт. Я встречал организации, требовавшие обязательный формальный просмотр кода перед отправкой в продакшен. Часто весь просмотр делает архитектор или ведущий разработчик. Такое положение вещей закреплено формально, поэтому разработчикам приходится его принять. Возможно, некоторым организациям действительно нужно соблюдать столь строгие правила, но большинству это не надо. Более того, для них такая практика оказывается контрпродуктивной. Те, чей код просматривают, могут себя чувствовать проверяемыми. А проверяющим нужно время для самой проверки и время, чтобы постоянно быть в курсе всех подробностей системы. Проверяющие очень быстро могут стать узким местом всего процесса, что приведет к его деградации.

Вместо исправления ошибок в коде целью ревью кода должно быть распределение знания и установление общих рекомендаций кодирования. Выкладывание своего кода в доступ для всех остальных членов команды «включает» коллективное владение кодом. Пусть взятый случайно программист «пробежится» по коду остальной команды. Вместо поиска ошибок акцент нужно при этом делать на попытке понять, что и как делает просматриваемый код.

Будьте мягкими во время просмотра. Обеспечьте, чтобы замечания были конструктивными, не едкими. Представьте различные роли просмотра, чтобы избежать влияния иерархической структуры команды на процесс ревью кода. Примеры ролей – просмотр документации, просмотр системы исключений, просмотр функциональности. Такой подход позволяет распределить груз ревью кода по всем членам команды.

Выделите отдельный *«день просмотра кода»* в неделю. Тратьте пару часов на обсуждения просмотра. Обеспечьте равномерную ротацию ролей просматривающих. Меняйте роли на каждом обсуждении ревью. *Вовлекайте в просмотр новичков*. У них может не быть достаточно опыта, но при этом у них будет «свежий взгляд», что тоже имеет свою ценность. *Вовлекайте в процесс экспертов*. Они смогут быстрее и точнее

найти проблемные места. Процесс просмотра будет идти быстрее, если в команде будет установленный стиль кодирования, проверяемый автоматически. Тогда обсуждение стиля не будет выноситься на обсуждения.

Самый главный залог успеха – сделать процесс просмотра привлекательным. Если процесс создает проблемы или вызывает негатив, то будет сложно кого-либо мотивировать его проводить. Сделайте его неформальным. Сделайте основной целью просмотра распределение знаний между членами команды. Вместо сарказма старайтесь быть позитивными.

Автор оригинала - Mattias Karlsson

Думаете, это никто не увидит?

(В оригинале - Don't Be Cute with Your Test Data)

> *Было уже поздно. Мне надо было ввести тестовые данные, чтобы посмотреть, как будет выглядеть разметка страницы.*
>
> *В качестве имен пользователей я внес имена членов группы The Clash. Названия фирм? Для них подойдут названия песен группы Sex Pistols.*
>
> *Для идентификаторов биржевых аппаратов нужно было ввести четырехбуквенные слова в верхнем регистре, и конечно же, я ввел те самые четырехбуквенные слова (avl: в английском языке основные ругательства четырехбуквенные).*
>
> *Это все выглядело безобидно. Всего лишь небольшое развлечение для меня и возможно для разработчиков, кто успеет это увидеть до того, как завтра в систему будут внесены реальные данные.*
>
> *А на следующее утро менеджер проекта сделал несколько скриншотов для презентации.*

История программирования изобилует историями, в которых разработчики делали что-то, что «никто никогда не увидит», и что случайно становилось видимым в самый неподходящий момент.

Варианты утечек могут быть разными, но когда такая утечка происходит, она может оказаться фатальной для человека, команды или компании. Примеры не надо долго искать.

- Во время важного совещания клиент нажимает на кнопку с надписью «Не реализовано». Появляется сообщение «Не нажимай больше сюда, идиот!».
- Программисту, сопровождающему старую систему, дают задание добавить сообщения об ошибках, и он решает задействовать уже имеющийся, но неиспользуемый механизм. И пользователи начинают получать сообщения вроде «О боже, отказ базы данных, Бетмен!».
- Кто-то перепутал реальный и тестовый интерфейсы и внес что-нибудь веселое, в результате в реальном онлайн-магазине в продаже появляется «Массажер в виде Билла Гейтса».

Старая пословица говорит, что «Ложь может обойти полмира пока правда будет надевать ботинки», а в современном мире информация о вашей оплошности может распространиться через живой журнал, твиттер или одноклассники еще до того, как разработчики о ней узнают.

И даже исходные коды не защищены от пристального исследования. В 2004 году архив с исходными кодами Windows 2000 попал в открытый доступ, и нашлись люди, которые детально покопались в нем на предмет ругательств, оскорблений и других интересных комментариев (С тех пор я порой использую комментарий «Ужасный Страшный Отвратительный Мерзкий ХАК» в подходящих местах).

Подводя итог: когда вы что-либо пишете в коде, будь то комментарий, запись в логе, диалог или тестовые данные, всегда задавайте себе вопрос, что произойдет, если это увидит кто-нибудь еще. Возможно, это поможет вам не попадать в ситуации, в которой вам придется краснеть и желать провалиться сквозь землю.

Автор оригинала - Rod Begbie

Единственный исполняемый файл

(В оригинале - One Binary)

Мне встречались проекты, в которых процесс сборки модифицировал исходный код с целью получить различные исполняемые файлы для различного окружения. Такой подход все только усложняет и добавляет риск того, что версии окажутся рассогласованными между собой. Такой подход приводит к получению нескольких практически идентичных копий ПО, каждую из которых нужно установить в корректное окружение. Слишком много движущихся частей в механизме повышают вероятность того, что что-нибудь где-нибудь пойдет не так.

Однажды я работал в команде, где изменение каждого свойства запускало полный процесс сборки, в результате тестерам значительную часть времени приходилось ждать, даже в случае незначительных изменений (и да, процесс сборки был, конечно же, очень долгим). На другом проекте, на котором я работал, было правило запуска процесса сборки с нуля для получения финальной версии – релиза, что означало то, что не было гарантии того, что клиент получал то же самое, что тестировали тестеры.

Правило же здесь простое: создавать единственную версию исполняемых файлов, с возможностью идентификации этой версии и продвижения ее по «линии жизни» проекта. А специфику среды сохранять непосредственно в среде окружения (в файле или папке, например)

Если у вас процесс сборки изменяет исходный код или сохраняет установки среды непосредственно в коде, это означает лишь то, что никто на этапе дизайне не продумал того, как разделить ядро приложения от платформо-зависимых деталей. Или даже хуже – команда знала, что и как надо делать, но приоритеты были расставлены так, что сделано этого не было.

Конечно же, бывают и исключения. Возможно, вам нужно будет делать сборку для различных окружений, имеющих очень серьезные отличия и ограничения. Однако эти исключения не применимы в большинстве случаев из серии «извлечь из базы, показать на экране и записать назад в базу». Или же вам приходится мириться с доставшимися в наследство проблемами, которые сложно исправить прямо сейчас. В таком случае вам нужно последовательно двигаться вперед, небольшими шагами, но начать движение надо как можно раньше.

И еще одна важная деталь. Сохраняйте установки среды окружения в системе контроля версий наравне с кодом. Нет ничего хуже, чем повредить конфигурацию и не быть в состоянии посмотреть, какое именно изменение все сломало. Информацию об

окружении лучше помещать отдельно, поскольку она меняется с другой частотой и по другим причинам по сравнению с кодом. Некоторые команды используют распределенные системы контроля версий для этого (подобные bazaar и git), поскольку в них проще поместить изменения непосредственно из рабочего окружения (что рано или поздно, но неизбежно случается) назад в репозиторий.

Автор оригинала - Steve Freeman

Заботьтесь о коде

(В оригинале - You Gotta Care about the Code)

Не надо быть Шерлоком Холмсом, чтобы заметить, что хорошие программисты пишут хороший код. А плохие – нет. Они пишут нечто монстроподобное, и это потом приходится разгребать остальным. Вы, конечно же, хотите писать код хорошо, не так ли?

Хороший код не появляется из воздуха. И не появляется благодаря правильному положению планет на небе. Над хорошим кодом надо работать. Тяжело работать. И чтобы получить хороший код, надо стараться получить хороший код.

Чтобы стать хорошим программистом, недостаточно быть хорошим техническим специалистом. Я видел много программистов, способных реализовать алгоритмы очень высокой сложности и знающих стандарты языков назубок, но при этом писавших полный ужас, который было больно читать, больно использовать и особенно больно изменять. И я видел более скромных программистов, не уходивших далеко от простых конструкций, но при этом пишущих элегантные программы, с которыми приятно иметь дело в дальнейшем.

Основываясь на своем опыте в индустрии ПО, я пришел к выводу, что главное отличие хорошего программиста от просто программиста – это *отношение*. Хорошее программирование возникает из желания писать как можно лучше, учитывая ограничения реального мира и давление индустрии.

Как известно, дорога в ад вымощена благими намерениями. Чтобы стать хорошим программистом, вам нужно перерасти благие намерения и начать реально *заботиться* о своем коде – поощрять позитивные перспективы и развивать здоровое отношение. Замечательный код отшлифован с тщательностью высококлассного ремесленника, а не бездумно напедален небрежным программистом или же магически наколдован гуру от программирования.

Итак, вы хотите быть хорошим программистом и писать хороший код. Значит, вам придется делать следующие вещи.

- В любой ситуации вы не должны соглашаться оставлять код, который лишь кажется, что работает. Вы должны пытаться получить полностью корректный и работоспособный код (вместе с хорошими тестами, чтобы это подтвердить).
- Вы должны писать понятный код (который другие программисты смогут легко взять и разобраться в нем), сопровождаемый код (который вы или другой

программист легко сможет менять в будущем) и работающий код (вы должны как можно сильнее убедиться, что вы полностью решили задачу, а не только сделали вид, что программа работает).

- Вы хорошо работаете вместе с другими. Мало кто работает изолированно. Большинство работают в команде, будь то компания или проект с открытым кодом. Вы считаетесь с другими программистами и пишете код так, чтобы его могли читать. Вы стараетесь, чтобы вся команда сделала продукт как можно лучше вместо того, чтобы стараться выглядеть умнее остальных.
- Каждый раз, когда вам приходится менять код, вы немного его улучшаете (делаете его более структурированным, более оттестированным, более понятным…)
- Вы все время изучаете новые языки, идиомы и техники. Но применяете их только тогда, когда это необходимо.

К счастью, вы сейчас читаете эту коллекцию советов именно потому, что вы хотите стать лучше в своей практике программирования. Это вам интересно. Это ваша страсть. Программируйте с интересом. Радуйтесь найденным решениям сложных задач. Делайте ПО, за которое можно испытывать гордость.

Автор оригинала - Pete Goodliffe

Золотое правило дизайна API

(В оригинале - The Golden Rule of API Design)

Проектирование API – это сложно, особенно больших API. Если вы проектируете API, который будут использовать сотни или тысячи людей, то вы должны подумать о том, как вы будете его менять в будущем и не приведут ли эти изменения к тому, что код ваших клиентов перестанет работать. Кроме этого, вам нужно подумать, как пользователи вашего API будут влиять на вас. Если один из ваших API классов использует собственный метод, то помните, что пользователь может унаследовать свой класс от вашего, перекрыв этот метод, и это может привести к катастрофе. Вы не сможете поменять этот метод, потому что кто-то из ваших пользователей его переделал. И ваша внутренняя реализация начинает зависеть от ваших пользователей.

Разработчики API решают эту проблему по-разному, но самый простой способ – блокировка API. Так, на Java вы можете сделать большинство ваших классов final. В C# таким ключевым словом будет sealed. Независимо от языка вы можете захотеть, чтобы API предоставлялось через singleton или через статические фабричные методы с целью защитить себя от переопределения поведения и использования вашего кода, которое может вас ограничить в будущем. Это все звучит логично, но реально ли это все сделать?

В течении последнего десятилетия мы постепенно осознали всю важность модульного тестирования, но это знание еще не полностью распространилось в отрасли. Подтверждение этому вокруг нас. Возьмите любой непротестированный класс, использующий какой-нибудь сторонний API, и попытайтесь написать для него юнит-тест. Практически наверняка вы столкнетесь с проблемами. Вы обнаружите, что код, использующий API, буквально приклеен к этому API. Окажется практически невозможным отделить API так, чтобы отследить, как код взаимодействует с ним и перехватить возвращаемые значения для проверки.

Со временем состояние может улучшиться, но только если мы в процессе проектирования API начнем смотреть на тестирование как на реальный пример его использования. И речь здесь не о тестировании только нашего кода. Здесь начинается Золотое Правило Проектирования API: «Недостаточно написать тесты для разрабатываемого вами API, нужно написать тесты для кода, использующего ваш API». Когда вы это проделаете, вы на себе испытаете все те проблемы, с которыми столкнутся те, кто будет использовать ваш API и писать тесты для своего кода.

Не существует простого способа сделать API удобным для тестирования кода тех, кто этот API будет использовать. `Static`, `final` и `sealed` по сути неплохие конструкции и могут использоваться при необходимости. Однако кроме этого нужно иметь в виду процедуру тестирования и пройти через нее самому. Испытав это на себе, вы сможете учитывать ее тонкости при проектировании.

Автор оригинала - Michael Feathers

Изучайте гуманитарные науки

(В оригинале - Read the Humanities)

Практически в любом проекте разработки ПО (кроме самых маленьких) люди работают вместе с другими людьми. Во всех областях, кроме наиболее абстрактных исследований, люди пишут ПО для людей, чтобы помочь им в достижении своих целей. Люди пишут ПО для людей. Этот бизнес связан с людьми. К сожалению, в процессе обучения программистов очень мало внимания уделяется аспекту взаимодействия с людьми – коллегами и заказчиками. К счастью, есть множество исследований, которые могут в этом помочь.

Например, Людвиг Витгенштейн (Ludwig Wittgenstein) выдвинул очень хорошее предположение в работе «Философские Исследования» (Philosophical Investigations), что ни один язык, используемый нами, не может и не является универсальны форматом для передачи мысли, идеи или образа из одной головы в другую. В связи с чем нам уже нужно быть готовым к непониманию в процессе выяснения требований к системе. Витгенштейн также показал, что наша способность понимать друг друга – это следствие не общих определений, а общего опыта, образа жизни. Возможно, это одна из причин, почему программисты, погруженные в проблемы предметной области, работают лучше, чем те, кто от предметной области дистанцируется.

Лакофф и Джонсон (Lakoff and Johnson) представили нам каталог метафор, которыми мы живем (Metaphors We Live By), предполагая, что язык в значительной мере метафоричен, и эти метафоры дают нам шанс взглянуть на то, как мы воспринимаем мир. Даже вполне конкретный термин, как например, «денежный поток» (cash flow), используемый при обсуждении финансовых систем, может быть рассмотрен с метафорической точки зрения: «деньги – как вода». Как эта метафора может повлиять на наше представление системы, работающей с деньгами? Или же часто использующееся выражение «стек протоколов» с верхним и нижним уровнем – оно тоже крайне метафорично: «наверху» находится пользователь, а «внизу» - технология. Это влияет на то, как мы видим структуру проектируемой системы.

Мартин Хайдеггер (Martin Heidegger) изучил то, как люди пользуются инструментами. Программисты создают инструменты, и поэтому для них инструменты находятся в зоне их интересов. Однако для пользователя, как показал Хайдеггер в трактате «Бытие и время» (Being and Time), инструмент становится невидимым, незаметным. Инструмент начинает интересовать пользователя только тогда, когда перестает работать или работает неправильно. И об этой разнице стоит помнить, обсуждая аспекты удобства использования

Элеанор Рош (Eleanor Rosch) пересмотрела модель категорий Аристотеля, согласно которой мы выстраиваем свое понимание мира. Когда программист спрашивает пользователя о требованиях к системе, он старается получить ответ, построенный на предикатах (Предикат - языковое выражение, обозначающее какое-то свойство или отношение). Это весьма распространенная практика, поскольку предикат потом легко становится атрибутом класса или колонкой в таблице. Такой подход – строгий и аккуратный, но к сожалению, как показала Рош (в Natural Categories) и более поздних работах, люди представляют мир совершенно по-другому. Их понимание мира основано на примерах. Некоторые примеры могут быть лучше других, в результате давая неоднозначные, пересекающиеся категории со сложной внутренней структурой. И поэтому, настаивая на ответах в модели Аристотеля, мы не можем задать пользователю правильных вопросов о том, как он представляет мир, и нам будет сложно придти к общему пониманию.

Автор оригинала - Keith Braithwaite

Изучайте другие языки программирования

(В оригинале - Know Well More than Two Programming Languages)

Уже достаточно долго известен тот факт, что уровень мастерства программиста прямо пропорционален количеству различных парадигм программирования, которыми он владеет. Причем не просто знает о их существовании, а может их использовать в работе.

Любой программист начинает с изучения одного языка программирования. И этот язык оказывает преобладающее воздействие на то, как этот программист видит разработку ПО. Неважно, сколько лет он будет использовать этот язык, он все равно будет знать только этот язык и будет, так сказать, думать на этом языке.

Программист, изучающий другой язык, столкнется с проблемами, особенно если новый язык обладает другой моделью вычислений. C, Pascal, Fortran – все они используют одну и ту же модель. Перейти с Fortran на C проблем практически не составит. Переход от C к C++ потребует гораздо больше усилий, поскольку C++ уже имеет фундаментальные отличия. Переход от C++ к Haskell потребует еще большего изменения взглядов и как следствие, будет еще более сложен. А переход от C к Prolog будет представлять сложности почти наверняка.

Мы легко можем перечислить известные парадигмы вычислений – процедурная, объектно-ориентированная, функциональная, логическая и т.п. И переход между ними – наиболее сложная задача для программиста.

Что же эти сложности приносят положительного? Они заставляют нас думать о реализации алгоритмов, идиомах и паттернах в зависимости от используемой реализации. Основа экспертности – «перекрестное опыление». Идиомы одного языка могут отсутствовать в другом. И пытаясь перенести идиомы из одного языка в другой, мы обучаемся обоим языкам, а также способам решения проблем.

«Перекрестное опыление» в программировании производит огромный эффект. Один из самых характерных примеров – все возрастающее использование декларативных моделей в системах, реализованных на императивных языках. Каждый, кто погружался в функциональное программирование, легко применит декларативный подход даже при использовании языка вроде C. Применение декларативных

принципов приводит к более коротким и ясным программам. C++, например, использует это в своей ключевой технологии обобщенного программирования, не могущей обойтись без декларативной модели выражений.

Вывод из всего этого такой, что каждому программисту следует овладеть минимум двумя парадигмами, а еще лучше – всеми вышеперечисленными. Программисты должны быть заинтересованы в изучении новых языков, особенно из еще неизученных парадигм. Даже если повседневная работа требует применения только одного языка, не стоит недооценивать положительный эффект «переопыления» программиста идеями из другой парадигмы. Работодателям стоит иметь это в виду, закладывая в бюджет тренинги по языкам, не использующимся в настоящий момент, с целью повысить эффективность используемых.

Конечно же, недельного курса будет недостаточно, чтобы полностью выучить новый язык. По-хорошему, потребуется несколько месяцев использования, чтобы получить ощутимый опыт его использования. При этом важно понять использование идиом языка, а не только изучить его синтаксис и модель вычислений.

Автор оригинала - Russel Winder

Изучайте иностранные языки!

(В оригинале - Learn Foreign Languages)

Программистам приходится общаться. Много общаться.

В отдельные периоды жизни программист может общаться практически только с компьютером. Точнее, с программами, работающими в компьютере. Такое общение можно назвать выражением идей в понятной компьютеру форме. Программы – это идеи, воплощенные в реальности практически без участия материальной составляющей.

Программист должен свободно владеть машинным языком и абстракциями, с которыми этот язык имеет дело. Важно владеть множеством абстракций, иначе некоторые идеи будет очень сложно реализовать. Хорошие программисты должны находиться за пределами ежедневной рутины и быть готовыми изучать другие языки для других целей.

Кроме общения с компьютерами, программистам приходится общаться и с коллегами. Современные крупные проекты – скорее результат коллективных усилий, а не простого применения программистских техник. Важно понимать и быть способным объяснить гораздо больше, чем позволяют используемые компьютерные абстракции. Большинство лучших программистов, которых я знаю, очень хорошо владеют своим родным языком, а плюс к этому еще и несколькими другими. И дело здесь не только в коммуникации с другими – хорошее владение языком приводит к ясности мыслей, что необходимо для оперирования абстракциями. А именно этим программисты и занимаются чаще всего.

Кроме общения с компьютерами, собой и коллегами, приходится общаться и с другими участниками проекта с разной технической подготовкой или без нее. Эти участники занимаются тестированием, продажами, это также пользователи вашего продукта и его заказчики. Вам необходимо понимать их всех, что практически невозможно, если вы не владеете языком их предметной области. Вам будет казаться, что вы все поняли, однако это будет не так.

Если вы говорите с бухгалтером, вам нужны базовые знания о бухучете. Если вы общаетесь с адвокатом, вам нужно знать язык, на котором общаются юристы. Языком предметной области проекта обязательно должен владеть хоть кто-нибудь в команде, и идеально, если это – программисты. Ведь именно они воплощают идеи в жизнь.

В конце концов, жизнь – это не только программирование. Изучить другой язык – все равно что получить вторую душу. Вам пригодится знание иностранных языков и в вашей обычной жизни вне программирования. Чтобы знать, когда лучше слушать, чем говорить. Чтобы знать, что для общения почти не надо слов.

Автор оригинала - Klaus Marquardt

Изучите ограничения

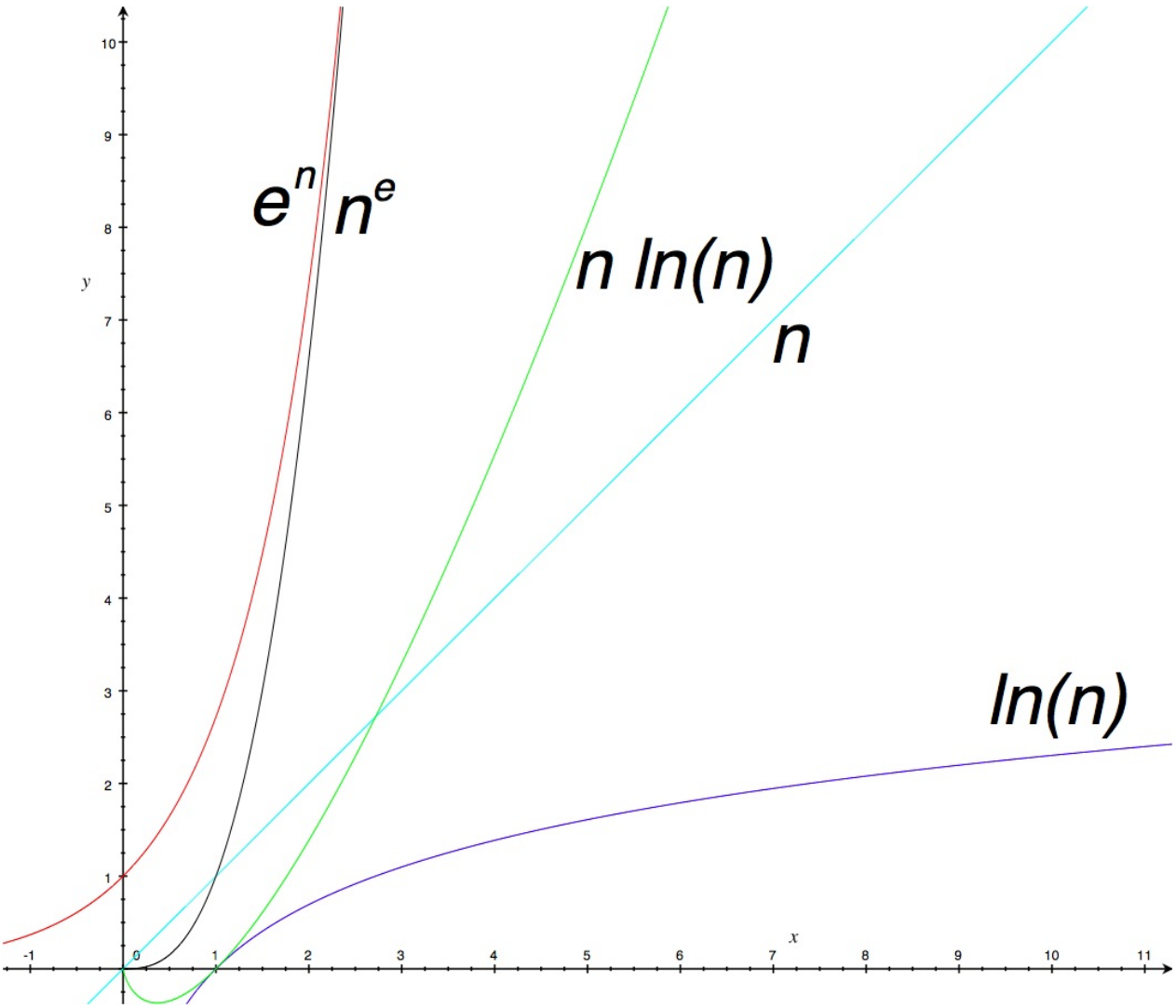
(В оригинале - Know Your Limits)

Вы имеете дело с ограниченными ресурсами. У вас есть лишь ограниченное количество времени и денег, чтобы сделать этот проект, включая время и деньги на получение знаний, умений и современные инструменты. Вы можете работать с ограниченной интенсивностью, скоростью и эффективностью, и так далее. Ваши инструменты имеют ограниченные возможности. Ваши компьютеры обладают ограниченной производительностью. Вам приходится принимать эти ограничения во внимание.

Как же учитывать эти ограничения? Узнайте себя, узнайте своих людей, узнайте бюджет и узнайте все остальное. Будучи программистом, вам особенно важно знать о сложности ваших структур данных и алгоритмов, а также архитектуру и производительность ваших систем. Ваша задача – создать оптимальное сочетание программ с аппаратными системами.

Сложность определяется функцией *O(f(n))*, являющейся для n, равного размеру входных данных, приближенным значением требуемого объема памяти или времени по мере приближения n к бесконечности. Важные классы сложности включают *ln(n)*, *n*, *n ln(n)*, n^e, and e^n

Графическое представление наглядно показывает, что по мере роста n *O(ln(n))* значительно меньше, чем *O(n)* и *O(n ln(n))*, а n в свою очередь значительно меньше $O(n^e)$ и $O(e^n)$. Для доступных значений n можно выделить три класса сложности – константная, линейная и бесконечная.



Анализ сложности делается в терминах абстрактного компьютера, однако программы работают на реальных компьютерах. Современные компьютеры – это иерархия физических и виртуальных машин, включая исполняемые языки, операционные системы, процессоры, кэш, память, диски и сети. В таблице ниже приведены примерные ограничения на скорость доступа и размеры определенных структур.

	access time	**capacity**
register	< 1 ns	64b
cache line		64B
L1 cache	1 ns	64 KB
L2 cache	4 ns	8 MB
RAM	20 ns	32 GB
disk	10 ms	10 TB
LAN	20 ms	> 1 PB
internet	100 ms	> 1 ZB

Заметьте, что объемы и быстродействие отличаются на несколько порядков. Кэширование используется практически на всех уровнях, чтобы скрыть эти различия, однако оно работает лишь для предсказуемого доступа. При частых «промахах кэша» система может замедлиться в разы. Например, чтобы в случайном порядке прочитать все байты жесткого диска, может понадобиться 32 года. А чтобы случайно просмотреть всю оперативную память – 11 минут. Случайный доступ непредсказуем. Исходя из этого, следует помнить, что повторный доступ к уже использовавшимся элементам и последовательный доступ практически всегда работают эффективно.

Алгоритмы и структуры данных значительно отличаются по эффективности использования кэша. Например:

- Линейный поиск выполняет последовательный перебор, но требует O(n) сравнений
- Бинарный поиск в отсортированном массиве требует лишь O(log(n)) сравнений
- Поиск по дереву van Emde Boas также требует O(log(n)) сравнений и при этом эффективно использует кэш (cache-oblivious)

Элементы	**Время поиска (нс)**		
	линейный	**бинарный**	**vEB**
8	50	90	40
64	180	150	70
512	1200	230	100
4096	17000	320	160

И как же сделать выбор? Измеряя. В таблице показано время, требуемое для поиска 64-битного целого числа тремя методами в массивах различного размера. На моем компьютере линейный поиск наиболее выгоден для небольших массивов, однако существенно проигрывает при увеличении размера. Алгоритм van Emde Boas выигрывает всегда благодаря предсказуемому доступу к данным.

Автор оригинала - Greg Colvin

Изучите свой IDE

(В оригинале - Know Your IDE)

В восьмидесятых годах среда разработки представляла собой всего лишь слегка украшенные текстовые редакторы… и то, если повезет. Подсветка синтаксиса, к которой мы так привыкли сегодня, тогда была роскошью, не всем доступной. Утилиты форматирования были внешними. Как и отладчики, с помощью которых можно было отлаживать пошагово, но при помощи замысловатых сочетаний клавиш.

В течении девяностых годов компании стали замечать потенциал, которые они бы могли получить, обеспечив программистов удобными средствами разработки. Интегрированные среды разработки (IDE) объединили в себе средства редактирования, компиляции, отладки, автоформатирования и другие возможности. Тогда же стали популярными меню и мышь, что привело к тому, что запоминать сложные сочетания клавиш больше было не нужно. Было достаточно выбрать необходимое действие из меню.

В двадцать первом веке IDE настолько распространились, что стали распространяться бесплатно с целью выйти на этот рынок. Современные IDE обладают просто фантастическим набором функциональностей. Моей любимой является автоматический рефакторинг, особенно Extract Method, где можно выделить кусок кода и сделать его функцией. Инструмент сам определит необходимые параметры, которые необходимо передать, что значительно упрощает процесс изменения. Кроме этого, IDE может даже найти другие места в коде, которые можно заменить на вызов этой новой функции, и предложить их заменить тоже.

Еще одно замечательное свойство современных IDE – возможность задавать стиль кодирования. Например, некоторые программисты Java стали делать все параметры final (что мне кажется потерей времени). Однако, чтобы начать следовать их правилу, все, что мне нужно – это настроить в IDE выдавать предупреждение для всех не-final параметров. Правила стиля могут использоваться и для поиска возможных ошибок, таких как сравнение autoboxed-объектов.

К сожалению, современные IDE не требуют больших инвестиций в изучение. Когда я начинал программировать на C под Unix, я должен был потратить значительное количество времени на изучение редактора *vi* (из-за ступенчатого характера кривой изучения). Это потраченное время потом многократно окупилось. Я даже черновик этой статьи набирал в *vi*. Современные IDE имеют очень пологую кривую изучения, что приводит к тому, что мало кто изучает что-то еще, кроме самых базовых вещей.

Мои первые шаги по изучению IDE – это запоминание горячих клавиш. Нажать *Ctrl+Shift+I* гораздо быстрее, чем выбрать мышкой пункт меню, отрываясь от основной работы. Переключения внимания на выбор действия мышью из меню приводят к серьезному снижению производительности.

И наконец, как программисты мы всегда можем обратиться к проверенным временем потоковым утилитам Unix, чтобы сделать что-то с нашим кодом. Например, если во время ревью кода я замечу, что программисты одинаково назвали классы, то чтобы это проверить, мне надо всего лишь набрать следующее:

```
find . -name "*.java" | sed 's/.*\///' | sort | uniq -c | grep -v "^ *1 " | sort -r
```

Давайте же потратим немного времени, чтоб работать с IDE более эффективно.

Автор оригинала - Heinz Kabutz

Инкапсулируйте не только состояние, но и поведение

(В оригинале - Encapsulate Behavior, not Just State)

В теории систем инкапсуляция является одной из наиболее часто используемых конструкций, когда речь заходит о больших и сложных системах. В индустрии ПО ценность инкапсуляции также хорошо осознаваема. Инкапсуляция поддерживается языками программирования наравне с функциями, модулями и пакетами.

Модули и пакеты инкапсулируют большие объемы и системы, а классы и функции – более тонкие аспекты. В течении многих лет я сталкиваюсь с тем, что именно классы являются самым проблемным местом для правильного использования инкапсуляции. Весьма часто можно увидеть класс, в котором присутствует единственный метод длиной в 3000 строк, или же класс, в котором реализованы лишь set() и get() методы для каждого атрибута. Эти примеры показывают, что разработчики не до конца понимают объектно-ориентированную модель, теряя возможность использовать всю ее мощь.

Объект инкапсулирует и состояние, и поведение, причем поведение зависит от актуального состояния. Представьте себе объект «Дверь». У него будет четыре состояния: «Открыто», «Закрыто», «Открывается», «Закрывается». И у него будет две операции: «Открыть» и «Закрыть». В зависимости от состояния, методы «Открыть» и «Закрыть» будут работать по-разному.

Такое присущее объекту свойство делает процесс проектирования относительно простым. Он сжимается до двух простых задач: назначение и делегирование ответственности от объекта к объекту, включая протокол взаимодействия объектов.

То, как это работает на практике, проще всего показать на примере. Пусть, например, у нас есть три класса: Customer, Order и Item. Объект Customer – естественное место для инкапсулирования кредитного лимита и правила его проверки. Объект Order знает об ассоциированном Customer, и его метод addItem делегирует саму проверку ему, вызывая `customer.validateCredit(item.price())`. Если условие не выполняется, генерируется исключение и покупка не выполняется.

Менее опытный ООП-разработчик может сделать по-другому – собрать все бизнес-правила в отдельный объект `OrderManager` (или `OrderService`). В таком дизайне `Order`, `Customer`, и `Item` рассматриваются лишь как набор данных. Вся логика же

вынесена из них и сосредоточена в большом методе класса `OrderManager`, с большим количеством ветвей и условий внутри. Такие методы очень легко «сломать» и почти невозможно поддерживать. Причина? Нарушенная инкапсуляция.

Подведя итог: не нарушайте инкапсуляции и используйте свойства языка для ее поддержки.

Автор оригинала - Einar Landre

Используйте правильные алгоритмы и структуры данных

(В оригинале - Use the Right Algorithm and Data Structure)

> Крупный банк со множеством филиалов выдвинул претензию, что купленные для кассиров новые компьютеры работают слишком медленно. Это было во времена, когда интернет-банкинг еще не был распространен, и люди приходили в банк гораздо чаще. Медленные компьютеры приводили к появлению очередей и недовольству клиентов. Вскоре банк пригрозил разорвать контракт с поставщиком.
>
> Поставщик отправил в банк специалиста, чтобы тот нашел причину медленной работы. Он очень быстро обнаружил, что одна из программ расходует практически 100% процессорного времени. При помощи профайлера он нашел «виновную» функцию, в которой было следующее:
>
> ```
> for (i=0; i<strlen(s); ++i) {
>   if (... s[i] ...) ...
> }
> ```
>
> Строка s была в среднем длиной в тысячи символов. Код (написанный работниками банка) был исправлен, и жалобы на медленную работу больше не появлялись.

Не должен ли программист писать чуть лучше, чем использовать безо всякой надобности алгоритм квадратичной сложности? Каждый вызов strlen приводит к последовательному проходу через всю строку до нахождения нулевого завершающего байта. При этом строка никогда не меняется в процессе. Если один раз найти и сохранить длину строки, то можно легко предотвратить тысячи вызовов strlen:

```
n=strlen(s);
for (i=0; i<n; ++i) {
  if (... s[i] ...) ...
}
```

Практически все слышали фразу «сначала сделай так, чтобы работало, потом делай, чтобы работало быстро», предотвращающую ненужную оптимизацию. Однако приведенный выше пример скорее следует правилу «сначала сделай так, чтобы работало медленно».

С такой невнимательностью вы можете столкнуться достаточно часто. И здесь не только принцип «не изобретай колесо». Иногда новички начинают писать код не думая и случайно «изобретают» пузырьковую сортировку, чем могут даже гордиться.

Другая сторона выбора правильного алгоритма – выбор структуры данных. Использование связанного списка для хранения миллиона записей с необходимостью поиска вместо использования хеш-таблицы или двоичного дерева доставят много неприятных минут пользователю вашего ПО.

Программистам не надо изобретать колесо, они должны использовать существующие библиотеки где только возможно. Однако чтобы избежать проблем вроде описанного случая с банком, им нужно знать про алгоритмы и про их сложность. Что же делает современные текстовые редакторы столь же медленными, как и их аналоги из эпохи 80-х? Многие считают повторное использование самым главным. Однако прежде всего программист должен знать, что, когда и как использовать повторно. Чтобы это знать, ему нужно знать про предметную область, алгоритмы и структуры данных.

Хороший программист также должен знать, когда стоит использовать сложные алгоритмы. Например, если предметная область определяет, что в массиве никогда не будет более пяти элементов, то вы всегда будете сортировать максимум пять элементов. И в этом случае пузырьковая сортировка будет наиболее эффективным алгоритмом. Всему свое время и место.

Так что читайте умные книги и разбирайтесь в том, что там написано. А если вы действительно прочитали Дональда Кнута «Искусство программирования» и смогли найти ошибку у автора, то можете получить один из его шестнадцатеричных долларов ($2.56).

Автор оригинала - JC van Winkel

Используйте преимущества анализаторов кода

(В оригинале - Take Advantage of Code Analysis Tools)

Важность тестирования усиленно вбивается в головы программистам с самого начала их карьеры. В последние годы набирающие популярность технологии юнит-тестирования, test-driven и agile разработки показывают на рост интереса внедрить тестирование во все фазы цикла разработки. Однако, тестирование – лишь один из инструментов для повышения качества вашего кода.

Давным-давно, когда C был новинкой, процессорное время и место на диске представляли существенную ценность. Поэтому компиляторы того времени снижали количество проходов до минимума, жертвуя при этом некоторой частью семантического анализа. Это означало, что компилятор проверял только часть ошибок, которые могли быть обнаружены во время компиляции. И для компенсации этого Стивен Джонсон (Stephen Johnson) написал lint – программу, выполняющую статический анализ кода, «выброшенный» из компиляторов. Однако, статистические анализаторы кода заслужили славу большого количества ложно-положительных предупреждений и стилистических конвенций, необязательных для безусловного следования.

Сейчас ситуация с языками, компиляторами и статическими анализаторами сильно отличается от того, что было ранее. Практически каждый язык может похвастаться хотя бы одной программой проверки на соответствие стилю и поиска коварных сложнообнаруживаемых ошибок вроде возможной разадресации нулевого указателя. Более продвинутые программы, вроде Splint для C или Pylint для Pyton, имеют возможности настройки того, какие ошибки и предупреждения программа будет игнорировать, как через конфигурационный файл, так и через опции командной строки. Splint даже позволяет использовать специальные комментарии, поясняющие ему, как работает ваша программа.

Если же вы «застряли» в поиске ошибки, которую не обнаруживает ни компилятор, ни IDE, ни какой-либо известный статический анализатор, то вы всегда можете сделать свой собственный анализатор. Это не так сложно, как может казаться. Большинство языков публикуют свое абстрактное синтаксическое дерево и компилятор как часть стандартной библиотеки. Знание потаенных уголков стандартной библиотеки очень помогает, поскольку там часто можно найти спрятанные жемчужины, могущие сильно помочь статическому анализу и динамическому тестированию. Например, в

стандартной библиотеке Python есть дизассемблер, который может выдать вам байт-код, используемый для генерации скомпилированного кода. С первого взгляда выглядит как что-то непонятное и пригодное только для авторов компилятора Python, однако на практике оказывается весьма полезным инструментом. Эта библиотека может дизассемблировать стек вызовов, точно указав, какая инструкция байт-кода вызвала последнее необработанное исключение.

Так что не позволяйте тестированию быть единственным инструментом повышения качества вашего ПО – используйте возможности программ-анализаторов, а при необходимости не бойтесь написать свой собственный анализатор кода.

Автор оригинала - Sarah Mount

Используйте типы из вашей предметной области

(В оригинале - Prefer Domain-Specific Types to Primitive Types)

23-го сентября 1999 года орбитальный комплекс ценой в 327.6 миллиона долларов был потерян при выходе на орбиту Марса. Виной всему стала ошибка в программном обеспечении, позже получившая классификацию «перепутанная метрика». Наземная часть системы управления использовала британскую систему (где сила измерялась в фунтах силы), а компьютер аппарата – метрическую (в ньютонах), в результате мощность маневровых двигателей была занижена в 4.45 раза. (Подробнее об этом можно прочитать тут или более подробно и на английском тут)

Этот и множество других примеров ошибок могли бы быть предотвращены, если бы использовался более строгий механизм типов предметной области. Это также объясняет множество свойств языка Ада, одной из основных целей разработки которого была реализация встраиваемых систем высокой надежности (safety-critical). Язык Ада использует строгую типизацию с проверкой на этапе компиляции как встроенных типов, так и типов, определенных пользователем.

```
type Velocity_In_Knots is new Float range 0.0 .. 500.00;

type Distance_In_Nautical_Miles is new Float range 0.0 .. 3000.00;

Velocity: Velocity_In_Knots;

Distance: Distance_In_Nautical_Miles;

Some_Number: Float;

Some_Number:= Distance + Velocity; -- В этом месте компилятор выдаст ошибку типизации.
```

Разработка приложений в менее критичных областях тоже может воспользоваться этим преимуществом определяемых типов вместо использования встроенных вроде string или float. В Java, C++, Python и других языках абстрактный тип – это класс. Использование классов вроде `Velocity_In_Knots` и `Distance_In_Nautical_Miles` значительно повышает качество кода:

1. код становится более читаемым, поскольку явно указываются значения из предметной области в отличие от «общих» типов Float или String;
2. код становится более тестируемым;

3. код способствует повторному использованию.

Такой подход одинаково хорошо подходит как для языков со статической типизацией, так и для языков с динамической. Единственная разница лишь в том, что при статической типизации разработчик получит подсказку сразу от компилятора, а при динамической для обнаружения подобных ошибок потребуется использовать юнит-тестирование.

Мораль – используйте типы из предметной области с целью разработки качественного программного обеспечения.

Автор оригинала - Einar Landre

Когда программисты и тестеры объединяются

(В оригинале - When Programmers and Testers Collaborate)

Когда программисты и тестеры объединяются, происходит что-то волшебное. Меньше времени тратится на пинание баг-репортов по системе трекинга ошибок. Меньше времени убивается на выяснение того, ошибка это или новая функциональность. И при этом больше времени уходит на разработку качественного софта, удовлетворяющего требованиям заказчика. Для начала такого сотрудничества еще до начала кодирования есть много причин.

Тестеры могут помочь заказчику написать и автоматизировать приемочные тесты на языке их предметной области с использованием инструментов вроде FIT (Framework for Intergrated Tests, платформа для интегрированных тестов). Когда эти тесты программисты получат еще до начала написания кода, они могут попрактиковаться в разработке, ведомой тестами (Test Driven Development). Программисты запускают тесты, после чего пишут код так, чтобы тесты успешно проходили. Эти тесты могут стать частью набора регрессионных тестов. В случае подобного сотрудничества функциональные тесты оказываются написанными достаточно рано, оставляя время на тестирование граничных условий или же тестирование основных сценариев работы.

Можно пойти еще дальше. У меня как у тестера обычно есть много идей еще до того, как программисты начнут писать код. И когда я спрашиваю программистов о том, нет ли у них каких-нибудь идей, обычно они дают мне информацию, позволяющую лучше покрыть тестируемую функциональность или же избежать траты времени на ненужные тесты. Часто в ходе такого диалога удается предотвратить проблемы, потому что тестирование проясняет многое еще на этапе идей. Например, однажды мои тесты, отданные программисту, подразумевали, что должен работать нечеткий поиск. Однако программист был твердо уверен, что нужно реализовывать только поиск по целым словам. Нам пришлось встречаться с заказчиком и прояснять требования еще до начала кодирования. В результате дефект был предотвращен, что привело к значительной экономии времени.

Программисты могут помогать тестерам автоматизировать тесты. Программисты лучше знают практики кодирования и могут помочь написать стабильные тестовые сценарии, работающие для всей команды. Я часто видел, как автоматизация тестирования проваливалась лишь потому, что она была неудачно спроектирована.

Тесты писались с целью протестировать слишком много, или же тестеры не до конца понимали технологию, чтобы писать тесты независимыми. Тестеры часто являются узким местом, поэтому в помощи от программистов есть смысл. Работа с тестерами с целью понять, что может быть протестировано на ранних стадиях, даст программистам возможность получить раннюю обратную связь, что в свою очередь поможет им написать более качественный и стабильный код.

Как только тестеры перестанут думать, что их задача – это лишь «сломать» ПО и найти ошибки в коде программистов, программисты перестанут думать, что тестеры здесь лишь чтобы доставить им проблемы. В результате все становятся более открытыми к сотрудничеству. Как только программисты осознают, что они отвечают за качество своего кода, тестирование станет естественным процессом, и команда вместе сможет автоматизировать больше регрессионных тестов. Так начинается магия успешной командной работы.

Автор оригинала - Janet Gregory

Комментируйте лишь то, что не ясно из кода

(В оригинале - Comment Only What the Code Cannot Say)

Различий между теорией и практикой гораздо больше на практике, чем в терории – это как нельзя более актуально применительно к комментариям. В теории идея комментариев заключается в том, чтобы «объяснить то, что происходит, дать больше деталей». Чем же желание объяснить может быть плохим? Однако на практике комментарии часто начинают приносить вред, а не пользу. Как и в любом другом деле, написание хороших комментариев – это умение. И значительная часть умения состоит в том, чтобы знать, когда их вообще писать не надо.

Если в коде есть ошибки, то компилятор, интерпретатор или другой подобный инструмент это сразу же обнаружат. Если ошибки есть на функциональном уровне, то ревью, статический анализ, тестирование и эксплуатация рано или поздно будут приводить к их обнаружению и устранению. А комментарии? В книге *“The Elements of Programming Style”* Kernighan и Plauger заметили, что «комментарий имеет нулевую или отрицательную ценность, если он неправильный». И такие комментарии часто сильно засоряют исходный код, оставаясь там долгое время, в отличие от ошибок в самом коде. При этом такие комментарии постоянно приводят к отвлечению внимания или даже к дезориентации того, кто этот код сопровождает.

А что, если комментарий не является технически неправильным, а лишь не приносит дополнительной ценности? Такие комментарии – это шум. Комментарий, повторяющий код, не дает читателю никакой дополнительной информации – повторение одного и того же сначала на языке программирования, а потом на человеческом не делает код более понятным. Закоментированный кусок кода не выполняется, поэтому не несет ничего интересного ни во время выполнения, ни для изучающего код. К тому же такой код очень быстро устаревает. Система контроля версий делает то, для чего предполагалось оставить закомментированный код, а именно отслеживание изменений, гораздо лучше.

Обилие «шумовых» и вообще неправильных комментариев в коде приводит лишь к игнорированию комментариев вообще. Программист легко может сделать что-нибудь, что может повлечь реальную проблему – игнорирование действительно важного комментария: спрятать все комментарии, настроить IDE так, чтоб комментарии отображались цветом фона, удалить все комментарии при помощи скрипта или еще что-нибудь подобное. И чтобы предотвратить такой сценарий, к комментариям надо

относиться так, словно это не комментарий, а код. Любой комментарий должен иметь свою собственную ценность для читателя, в противном случае его не надо писать вообще, удалить написанный или же переписать заново.

Что же считать ценностью? Комментарий должен сообщать что-то, чего код не сообщает и не может сообщить. Желание написать комментарий, содержимое которого может быть видно из кода – это сигнал к изменению стиля кода так, чтобы код говорил сам за себя. Вместо объяснения непонятных имен функций, классов или полей, просто переименуйте их. Вместо комментирования секций длинной функции разбейте ее на несколько мелких и назовите их адекватными именами. Лишь то, что вы на самом деле не можете «сказать» в коде, является кандидатом на комментарий.

Комментируйте лишь то, что код не может сообщить в принципе, а не то, что он не сообщает сейчас.

Автор оригинала - Kevlin Henney

Красота и простота

(В оригинале - Beauty Is in Simplicity)

Есть одна цитата, которую стоит знать каждому разработчику. А еще лучше эту цитату не просто знать, а все время помнить.

> *«Красота стиля, гармонии, грации и ритма определяется простотой»* - Платон.

Нам всем как разработчикам ПО стоит почаще пользоваться этой цитатой как руководством к действию.

Мы добиваемся того, чтобы наш код обладал следующими качествами:

- Читаемостью;
- Сопровождаемостью;
- Скоростью разработки;
- Красотой.

И Платон говорит нам, что простота – это то, что открывает дверь для всех этих свойств.

Что такое красивый код? Вопрос, возможно, достаточно субъективный. Восприятие красоты сильно зависит от индивидуального опыта, как впрочем и любое другое восприятие. Люди, выросшие в среде искусства, воспринимают прекрасное не так, как люди, выросшие в технической среде. Сторонники искусства видят красоту в коде, сравнивая программирование с художественным искусством, а сторонники науки больше говорят о симметрии и золотом сечении, пытаясь свести все к формулам. И мой опыт говорит, что основанием для аргументов обеих сторон является именно простота.

Подумайте об исходном коде, в котором вам пришлось разбираться. Если вам этого делать не приходилось, то прекратите читать эту статью и прямо сейчас найдите какой-нибудь open source проект для изучения. Серьезно! Я не шучу! Найдите в интернете что-нибудь на каком-нибудь языке программирования по вашему выбору, написанное каким-нибудь известным экспертом.

Вернулись? Отлично. На чем мы остановились? Ага, точно. Я обнаружил, что код, который я считаю красивым, обладает общими свойствами. Основное из которых – это простота. Я обнаружил, что каким бы сложным не был проект в целом, отдельные его части должны оставаться простыми. Простые объекты, решающие единственную

задачу, содержащие простые методы с понятными и информативными именами. Некоторые считают, что короткие методы по пять-десять строк кода – это перебор, и в некоторых языках достичь такого сложно, но я все равно считаю, что такая краткость – это желанная цель.

Основная идея в том, что красивый код – это простой код. Каждая отдельная часть остается простой с простыми зависимостями и связями с другими частями системы. Делая так, мы все время поддерживаем систему в сопровождаемом состоянии, просто написанную, легко тестируемую и позволяющую поддерживать высокую скорость разработки в течении всей жизни проекта.

Красота рождается из простоты.

Автор оригинала - Jørn Ølmheim

Мины замедленного действия

(В оригинале - The Road to Performance Is Littered with Dirty Code Bombs)

Часто, когда вам необходимо улучшить производительность системы, вам приходится вносить изменения в код. А когда необходимо менять код, то каждый переусложненный или излишне связанный фрагмент – мина замедленного действия, ждущая момента, чтобы пустить под откос вашу эффективность. Пока все идет гладко, вы достаточно точно можете предвидеть, когда вы закончите. Однако неожиданности «грязного» кода делают такое предвидение невозможным.

Предположим, вы нашли «узкое место» в коде. И для оптимизации решили заменить алгоритм нижележащего уровня. И допустим, вы оценили необходимое время часа в 3-4, о чем и доложили менеджеру. Однако после успешной замены алгоритма вы вдруг обнаруживаете, что перестала работать другая часть системы, зависящая от измененной. Возможно, вы предвидели и это (связность блоков на один шаг вполне можно предвидеть), но что, если после исправления этой части сломается что-нибудь еще? И чем далее от первоначального места будет продолжаться эта связность, тем ниже шансы, что вы ее обнаружите и учтете в оценке требуемого времени. И внезапно может оказаться, что реально вы потратите не 3-4 часа согласно первоначальной оценке, а 3-4 недели. Не такая уж и редкость наблюдать, как для завершения «быстрого» изменения в результате требуется несколько месяцев. И такая ситуация легко может сильно подорвать доверие к команде. Этого можно было бы избежать, будь у нас в руках инструмент, позволяющий оценить риск такого развития дел.

К счастью, существует множество способов контролировать степень и глубину связности и сложности кода. Метрики кода используются для подсчета тех или иных свойств кода. Значение этих метрик коррелирует с качеством кода. Для измерения связности применяются две метрики: fan-in и fan-out. Рассмотрим fan-out для отдельно взятого класса: эта метрика является количеством классов, на которые явно или неявно ссылается выбранный класс. Иными словами, это количество классов, которые должны быть откомпилированы прежде, чем откомпилируется ваш класс. Метрика fan-in, наоборот, показывает, сколько классов зависит от выбранного класса. Зная обе метрики, fan-in и fan-out, мы можем вычислить коэффициент нестабильности: $I = f_O / (f_I + f_O)$

Чем ближе значение этого коэффициента к 0, тем стабильнее код. Чем ближе значение к 1, тем код нестабильнее. Чем стабильнее код, тем проще его менять, и наоборот, чем код нестабильнее, тем больше в нем «мин замедленного действия». И цель рефакторинга – это уменьшение значения коэффициента нестабильности.

Используя эти метрики, помните и о здравом смысле. С математической точки зрения снизить коэффициент можно, повышая f_i, однако это будет означать появление классов с очень высоким значением f_i, которые сложно будет менять, не повреждая других связей. Без изменения fo вы на самом деле не снизите риска, так что придерживайтесь определенного баланса.

Недостатком метрик кода может быть то, что огромные массивы чисел, получающиеся на выходе систем подсчета метрик, могут остаться нерассмотренными. Метрики кода могут стать серьезным инструментом в борьбе за качественный код. Они могут помочь найти и обезвредить «мины замедленного действия» в коде еще до того, как они смогут нанести серьезный вред проекту.

Автор оригинала - Kirk Pepperdine

Миф о гуру

(В оригинале - The Guru Myth)

Каждый, кто проработал в ИТ достаточно долго, обязательно слышал вопрос вроде этого:

> *«У меня произошло такое-то исключение, в чем может быть проблема?»*

При этом спрашивающие обычно не считают нужным приложить стек вызовов, журнал ошибок или подробно описать контекст, в котором это происходит. Похоже, что они думают, будто вы находитесь на другом уровне, где ответы на вопросы возникают без подробного анализа обстоятельств события. Они думают, что вы гуру.

Такие вопросы ожидаемы от тех, кто далек от ИТ. Для них все эти системы кажутся почти магическими. А вот что меня беспокоит, так это то, что подобные вопросы все чаще появляются и в сообществе программистов. Похожие вопросы всплывают и в контексте проектирования ПО, вроде «Я проектирую складской учет, нужно ли мне использовать оптимистичную блокировку?» Самое смешное то, что часто те, кто спрашивают, более подготовлены к тому, чтобы дать ответ, чем те, у кого они это спрашивают. Задающий вопрос скорее всего знает контекст, знает требования и может прочитать о плюсах и минусах той или иной стратегии. И все равно они ожидают от вас адекватного ответа без знания контекста. Они ожидают магии!

Пришло время развенчать миф о гуру в индустрии ПО. Гуру – тоже люди. Они применяют логику и систематический анализ точно также, как и мы. Они используют интуицию. Даже самый лучший программист в мире может знать меньше о том, что вы сейчас делаете. И если кто-то кажется вам гуру, это из-за многолетней практики, обучения и усовершенствования мыслительного процесса. Гуру – это лишь умный человек с неослабевающим любопытством.

Конечно же, может существовать огромная разница в тех или иных умениях. Многие хакеры умнее, образованнее и продуктивнее, чем я когда-либо. И даже с учетом этого разоблачение мифа о гуру окажет позитивное воздействие. Например, работая с кем-то, кто способнее меня, я все равно буду стараться предоставить ему достаточно данных для того, чтобы он мог эффективно применить весь свой опыт. Развенчивание мифа о гуру также приведет к устранению воспринимаемого барьера для саморазвития. Вместо магического барьера я вижу среду, в которой я могу двигаться вперед.

И наконец, одна из самых больших проблем – умные люди, целенаправленно распространяют миф о гуру. Возможно, они делают это из-за эго или же для повышения собственной значимости в глазах клиента или работодателя. Однако такой подход может сделать умных людей менее ценными, поскольку они не будут способствовать развитию их напарников. Нам не нужны гуру. Нам нужны эксперты, желающие взрастить других экспертов в своей области. Нам всем хватит места.

Автор оригинала - Ryan Brush

Много данных? Используйте СУБД!

(В оригинале - Large Interconnected Data Belongs to a Database)

Если ваше приложение работает с большим объемом связанных друг с другом данных, то без каких-либо сомнений помещайте их в реляционную базу данных. В прошлом реляционные базы данных (РДБ, или СУБД) были дорогими, редкими, сложными и громоздкими. Сейчас это не так. СУБД стали широко распространены (скорее всего, на вашем компьютере установлена одна или две). Среди них есть системы с открытым кодом, такие как MySQL и PostgreSQL, так что цена тоже перестала быть критическим фактором. И даже больше, СУБД реального времени могут линковаться в виде библиотек непосредственно в приложение, так что вам даже не придется иметь дело с процедурой установки. Среди СУБД реального времени можно выделить SQLite и HSQLDB. Эти системы обладают очень высокой эффективностью.

Если объем ваших данных превышает объем оперативной памяти, то индексированная таблица в БД будет в разы эффективнее, чем размещение данных в памяти с использованием механизма виртуальной памяти. Современные БД легко масштабируются по мере надобности. Соблюдая определенные правила, вы легко можете перейти от встраиваемой СУБД к более продвинутой, если это понадобится. Или перейти с бесплатной БД с открытым кодом к коммерческой, имеющей техподдержку и более масштабируемой.

Как только вы освоите SQL, написание БД-приложений станет для вас простым и приятным занятием. После внесения нормализованных данных в БД вы легко можете извлекать их в различных комбинациях и подмножествах при помощи понятных SQL-запросов. Простая SQL-команда легко может выполнить весьма сложное изменение данных. А для однократных изменений вам вообще не понадобится писать код, достаточно будет запустить SQL-интерфейс и ввести команду там. В этом же интерфейсе вы можете экспериментировать с различными запросами – вместо аналогичного цикла «редактирование-компиляция-запуск».

Еще одно преимущество использования СУБД – возможность задания отношений между элементами данных. Вы можете задать ограничения целостности данных и тем самым избежать риска получения «висящих указателей» в случае некорректной работы с данными. Например, вы можете указать, что при удалении пользователя необходимо удалить и все посланные им сообщения.

Кроме этого, вы всегда можете легко создать связи между данными в базе, используя индексы. Вам не потребуется делать дорогой и сложный рефакторинг полей классов. К тому же СУБД позволяет организовать безопасный доступ к данным множеству пользователей. Это дает вам возможность перейти при необходимости к многопользовательскому режиму, а также реализовывать различные части приложения на наиболее подходящих платформах и языках. Например, вы можете написать XML back-end для веб-приложения на Java, скрипт аудита на Ruby, а интерфейс визуализации – на Processing.

И наконец, помните, что СУБД заботятся об оптимальном выполнении ваших SQL-команд, позволяя вам сконцентрироваться на функциональности приложения, а не на оптимизации алгоритмов. Продвинутые СУБД отлично работают на многоядерных процессорах. И по мере роста технологий будет расти и производительность вашего приложения.

Автор оригинала - Diomidis Spinellis

Мыслите состояниями

(В оригинале - Thinking in States)

Обычные люди часто испытывают проблемы, когда речь заходит о состояниях. Однажды утром я зашел в магазинчик подготовиться к очередному дню преобразования кофеина в строки кода. Поскольку я люблю кофе с молоком, а молоко как раз закончилось накануне, я хотел купить его, но к своему удивлению, не смог его найти. Тогда я обратился к продавцу и услышал в ответ «У нас совсем нет молока, извините».

Для программиста такая фраза звучит странно. Молоко или есть, или его нет. Когда молоко заканчивается, то далее его просто нет. Совсем нет или же нет самую чуточку — это одно и то же состояние. Возможно, продавец хотел таким способом сообщить мне, что с молоком действительно сложно в настоящее время, но для меня это значило лишь одно — сегодня я буду пить эспрессо.

Конечно, в повседневной жизни такое вольное отношение к состояниям не приводит ни к чему плохому. Но к сожалению, столь же вольное отношение встречается и среди программистов.

Представьте простой интернет-магазин, принимающий только кредитные карты, с классом `Order`, содержащим следующий метод:

```
public boolean isComplete() {
    return isPaid() && hasShipped();
}
```

Выглядит логично, не так ли? Однако если даже выражение красиво завернуто в метод вмето того, чтобы быть множество раз скопипасченым по всему коду, этого выражения не должно было вообще быть. То, что оно существует, показывает наличие проблемы. Почему? Потому что заказ не может быть доставлен, пока он не будет оплачен. Поэтому `hasShipped` не может стать true до того, как `isPaid` станет true, что делает выражение избыточным. Возможно, вы все равно захотите оставить метод `isComplete` для ясности кода, но он должен при этом выглядеть вот так:

```
public boolean isComplete() {
    return hasShipped();
}
```

Я много раз сталкивался как с пропущенными проверками, так и с избыточными. Этот пример крошечный, но если вы добавите отмену и повторную оплату, все станет гораздо сложнее, и необходимость правильной обработки состояний повысится. В данном случае, аказ может быть лишь в трех состояниях:

1. *Формируется:* можно добавлять или удалять покупки, нельзя доставлять;
2. *Оплачен:* нельзя добавлять или удалять покупки, можно доставлять;
3. *Доставлено:* заключительное состояние, никаких изменений.

Эти состояния важны, и вы должны проверять, что вы находитесь в ожидаемом состоянии перед выполнением операции, и что вы перемещаетесь в разрешенное состояние из текущего. Говоря иначе, вы должны аккуратно защищать объекты в правильных местах.

Но как начать думать состояниями? Оборачивание выражений в осмысленные методы — хорошее начало, но это лишь начало. Основа — это понимание машин состояний (конечных автоматов, state machines). Возможно (и скорее всего), вы уже успели забыть теорию цифровых автоматов из ваших институтских лекций, но в принципе, это и необязательно. Машины состояний — достаточно просты. Визуализируйте их чтобы упростить понимание. Испытайте ваш код на обнаружение разрешенных и запрещенных состояний и переходов. Изучите паттерн State. Когда почувствуете, что с паттерном разобрались, почитайте про контрактное программирование (Design by Contract). Это поможет вам гарантировать разрешенное состояние путем проверки входных данных и самого объекта на входе и выходе каждого метода.

Если вы обнаружили, что находитесь в неразрешенном состоянии, значит, произошла ошибка и вы рискуете потерять все данные, если не остановите выполнение. Если постоянные проверки состояния будет вносить слишком много шума, изучите возможность использования автоматической генерации кода или аспектного программирования (weaving), чтобы скрыть их. И независимо от того, какой механизм вы выберете, мышление категориями состояний поможет вам сделать ваш код более простым и надежным.

Автор оригинала - Niclas Nilsson

Наблюдайте за пользователями

(В оригинале - Ask "What Would the User Do?" (You Are not the User))

Все мы предполагаем, что другие люди думают так же, как и мы сами. Однако то не так. Психологи называют это склонностью к ложному согласию (false consensus bias). Когда люди думают или действуют не так, как мы, то мы часто считаем их в чем-то ненормальными.

Этот факт объясняет, почему программистам так сложно поставить себя на место пользователей. Пользователи не думают так, как программисты. Для начала, они используют компьютеры гораздо меньше времени. Они часто не знают и не хотят знать, как компьютеры работают. У них нет наработанных решений по решению проблем, что обычно для программистов. Они не знакомы с шаблонами и способами, которыми пользуются программисты для работы с пользовательским интерфейсом.

Лучший способ выяснить, как именно действуют пользователи, это понаблюдать за одним из них. Попросите пользователя выполнить какую-нибудь задачу при помощи ПО, похожего на разрабатываемое вами. Задача должна быть реальной. «Добавьте колонку с цифрами» - нормально. «Посчитайте свои расходы за прошлый месяц» - гораздо лучше. Избегайте слишком конкретных задач вроде «Вы можете выделить эти ячейки в ввести формулу суммы внизу?». Пусть пользователь говорит во время выполнения, не прерывайте его и не пытайтесь помочь. Спрашивайте себя «почему он делает так» и «почему он не делает так».

Первая вещь, которую вы заметите – это то, что пользователи делают большинство вещей одинаково. Они пробуют выполнить различные задачи одним и тем же способом, при этом делая одни и те же ошибки в одних и тех же местах. И вы должны проектировать вокруг их этого основного поведения. Это отличается от совещаний, на которых люди обсуждают вопросы из серии «А что если пользователь захочет сделать вот так?», и которые приводят к решениям, вызывающим замешательство пользователей. Наблюдение за пользователями устраняет это замешательство.

Вы можете увидеть, как пользователь зайдет в тупик. Когда вы в тупике, вы смотрите по сторонам. Когда пользователь в тупике, их фокус внимания сужается. Для них становится сложным найти решение где-то на экране. И это одна из причин, почему стандартный хелп – не самое лучшее решение. Если вам необходима текстовая помощь или инструкции, постарайтесь разместить их непосредственно вблизи зоны концентрации вашей задачи. Сужение фокуса внимания пользователя – одна из причин, почему всплывающие подсказки намного лучше, чем помощь в меню.

Пользователи часто делают вещи «как получится». Они находят работающий способ и потом везде используют только его. Поэтому лучше предоставить им один стандартный способ сделать что-то, чем предлагать два-три разных варианта.

Вы также обнаружите, что между тем, что пользователи говорят и тем, что они на самом деле делают, есть разница. И это проблема, поскольку обычно пользовательские требования формируются путем опроса. Поэтому лучший способ составить требования – это понаблюдать за пользователями. Потратить час на наблюдения будет гораздо более информативно, чем потратить день на распросы.

Автор оригинала - Giles Colborne

Научитесь пользоваться командной строкой

(В оригинале - Know How to Use Command-line Tools)

Сегодня множество инструментов разработки запакованы в Интегрированную Среду Разработки (IDE). Visual Studio и Eclipse – два популярных примера, хотя их конечно же намного больше. В IDE много хорошего. И это не только удобство использования. IDE помогает программисту не думать о множестве деталей, вовлеченных в процесс сборки.

Простота использования, однако, имеет и обратную сторону. Обычно инструмент легко использовать, потому что он принимает решения за вас, делая множество вещей автоматически, за сценой. Поэтому если IDE – это единственный используемый инструмент, то вы никогда до конца не узнаете, что же именно он делает. Вы нажимаете кнопку – и магическим образом в папке появляется исполняемый файл.

Работая с командной строкой, вы узнаете гораздо больше о том, что происходит во время сборки. Написание собственных make-файлов поможет вам понять все шаги (компиляция, ассемблирование, линковка и т.п.) сборки проекта в исполняемый файл. Экспериментирование с различными опциями этих шагов – не менее ценный опыт. Чтобы начать работу с командной строкой, можете взять open-source GCC или же то, что поставляется с вашим специфическим IDE. В конце концов, хорошо спроектированный IDE – это всего лишь графический интерфейс к набору утилит командной строки.

К тому же, некоторые вещи гораздо проще делать именно из командной строки. Например, поиск и замена при помощи grep и sed предоставляет гораздо больше возможностей, чем обычно присутствует в IDE. Командная строка также поддерживает скрипты, что позволяет автоматизировать задачи из серии ежедневной сборки по расписанию, создания нескольких версий проекта и запуск тестов. При помощи IDE такая автоматизация практически невозможна (или как минимум гораздо сложнее). Если вы никогда не заглядывали за границы IDE, вы могли и не догадаться о возможности подобной автоматизации.

Однако подождите… Разве IDE не предназначены для облегчения жизни разработчикам и повышения их продуктивности? Да, конечно. Предложение поработать с командной строкой не означает, что необходимо отказаться от IDE. Предполагается лишь заглянуть за завесу, чтобы понять, что именно делает IDE для вас. И лучший способ для этого – это изучить командную строку. Потом, когда вы опять

вернетесь к IDE, вы будете гораздо лучше понимать, что именно происходит и как вы можете контролировать процесс сборки. С другой стороны, научившись работать с командной строкой, возможно вы и сами не захотите возвращаться к IDE, прельстившись мощью и гибкостью командной строки.

Автор оригинала - Carroll Robinson

Начинайте с "да"

(В оригинале - Start from Yes)

Однажды я бродил по супермаркету, ища эдамаме (я лишь приблизительно знал, что это что-то овощное). Я не был уверен, где именно это искать – в овощной секции, в секции замороженных продуктов или среди консервов. В конце концов, я сдался и обратился за помощью к продавщице-консультанту. Но увы, она тоже этого не знала!

Сотрудница могла ответить множеством различных способов. Она могла просто ответить, что не знает, где это искать, или же предположить что-нибудь наугад, или же сказать, что такого у них вообще нет. Но вместо этого она восприняла мой вопрос как возможность помочь покупателю. Она позвонила кому-то еще, и через пару минут точно показала мне то, что я искал.

Сотрудница в данном случае исходила из предположения, что у них эта штука есть, и она сможет мне помочь ее найти. Она начала с «да» вместо того, чтобы начать с «нет».

Когда я впервые получил должность технического лидера, я ощущал эту работу как защиту своего замечательного ПО от потока идиотских требований, исходящего от бизнес-аналитиков и продуктных менеджеров. Я считал необходимым бороться практически со всеми поступающими предложениями.

Но в какой-то момент я вдруг прозрел и понял, что возможно, есть и другой способ работы, что привело к тому, что я вместо «нет»-начала все больше стал использовать «да»-начало. Фактически, я поверил в то, что именно так и должен работать технический лидер.

Это простое понимание радикально изменило мой подход к работе. Оказалось, что сказать «да» можно множеством способов. Когда кто-то говорит вам «Слушай, все, чего не хватает этому приложению – это сделать все окна круглыми и полупрозрачными», вы можете это отбросить как полную бессмыслицу. Однако гораздо лучше вместо этого спросить «А почему?». Часто у такого странного на первый взгляд желания есть вполне весомая причина. Возможно, что именно этого хочет от приложения новый крупный и перспективный клиент, готовый вот-вот подписать контракт.

Когда вы будете знать о причинах того или иного требования, вам откроются новые возможности. Часто требование можно удовлетворить, вообще ничего не делая: «Пользователь уже сейчас может загрузить себе скин с круглыми полупрозрачными

окнами».

Иногда у другого человека может быть идея, которая вам покажется несовместимой с вашим видением проекта. Мне в такой ситуации помогает задать вопрос «Почему?» самому себе. Иногда ответ на этот вопрос покажет, что ваша первая реакция была ошибочной. Если же нет, то вам следует вовлечь в принятие решения других ключевых людей. Помните, цель всего этого – сказать «да» другому человеку и постараться сделать так, чтобы это все заработало, как для него, так и для вас, и вашей команды.

Если вы можете высказать убедительное объяснение, почему предлагаемая штука не подходит для существующего продукта, то возможно, вам нужно обсудить, а действительно ли вы создаете требуемый продукт. Независимо от того, чем закончится это обсуждение, нужно сфокусироваться на том, что же продукт из себя представляет, а что – нет.

Начиная с «да», вы работаете вместе с коллегами, а не против них.

Автор оригинала - Alex Miller

Начните отладку процесса установки как можно раньше

(В оригинале - Deploy Early and Often)

Отладка процесса инсталляции часто откладывается на самый конец проекта. Иногда написание инсталляции делегируется инженеру, ответственному за релиз, который берется за это как за «неизбежное зло». Тесты и демонстрации делаются в окружении, при необходимости настроенным вручную, чтобы все работало. А в результате реальной информации о процессе не поступает вплоть до того момента, когда может быть уже слишком поздно.

Инсталляция – самая первая вещь, которую увидит заказчик. И простая процедура инсталляции – первый шаг к надежному (или как минимум легко отлаживаемому) процессу поставки. Установленное ПО – то, что будет использовать ваш заказчик. Но если в процедуре установки будут проблемы, то у заказчика возникнут вопросы к вам еще до того, как он начнет использовать ваше ПО.

Если вы начнете проект с реализации процесса установки, это даст вам время на его естественную эволюцию вместе с развитием самого продукта. А также поменять код продукта с целью упростить процесс инсталляции, если потребуется. Тестирование процедуры установки в различных средах также подтвердит (или не подтвердит) то, что вы не сделали неправильных предположений о среде разработки или установки.

Позднее начало работы над процедурой установки может привести к тому, что эта процедура окажется гораздо более сложной из-за неправильно сделанных предположений, внесенных в код приложения. То, что может казаться замечательной идеей, пока вы работаете из под IDE, может сильно усложнить последующий процесс установки продукта. И лучше об этом узнать как можно раньше.

Если «готовность к установке» не кажется чем-то суперсложным, пока приложение запускается на компьютере разработчика, то на самом деле вам потребуется значительное количество времени и усилий, чтобы обеспечить бесперебойную работу приложения на любом другом компьютере «с нуля». Если вы не беретесь за создание инсталляции по причине «это просто и быстро, всегда успеется», то все равно сделайте это вначале, поскольку это быстро и просто. Если же этот процесс слишком сложен или слишком неопределен, то действуйте так же, как вы действуете с основным приложением: экспериментируйте, меняйте и переделывайте процесс установки по мере надобности.

Процесс установки вашего ПО весьма существенен для эффективной работы вашего заказчика и вашей команды техподдержки. Поэтому тестируйте и меняйте его наравне с приложением.

Автор оригинала - Steve Berczuk

Не бойтесь что-нибудь сломать!

(В оригинале - Don't Be Afraid to Break Things)

Каждому, работающему в ИТ, приходилось работать на проекте, чей код был, мягко говоря, ненадежным. Любое изменение приводило к отказу в какой-нибудь другой, вообще независимой, части. Каждый раз при добавлении чего-нибудь основной целью было внести как можно меньше изменений, каждый раз затаив дыхание. Работать с таким ПО – все равно что играть в Дженгу в настоящем небоскребе, вытаскивая из конструкции несущие балки – рано или поздно такая игра закончится катастрофой.

Причина, по которой внесение изменений столь болезненно – то, что больна вся система. Системе нужен доктор, иначе ее состояние будет только ухудшаться. Вы уже знаете, что не так с вашей системой, но вы боитесь разбить яйцо, чтобы сделать яичницу. Опытный хирург знает, что при операции нужно будет резать по живому, но он также знает, что разрезы – это временно. Сразу после операции будет больно, но потом пациенту станет лучше, чем было до операции.

Не бойтесь своего кода. Кого интересует то, что какие-то части не будут работать, пока вы будете вносить изменения? Страх изменений и привел проект к столь запущеному состоянию. Инвестиции в рефакторинг окупятся многократно в течении жизни проекта. Ваша команда, поработав с «больной» системой, получила ценный опыт с точки зрения того, как система должна была быть написана правильно. Так возьмите и примените эти знания. Работать с системой, которую вы ненавидите – это не то, на что стоит тратить свое время.

Переопределите внутренние интерфейсы, переструктурируйте модули, переделайте копи-паст код и упростите дизайн, убрав ненужные зависимости. Вы можете серьезно снизить сложность, устраняя проблемные граничные случаи, часто являющиеся результатом некорректной комбинации параметров. Постепенно меняйте старую структуру на новую, постоянно тестируя результат. Попытка сделать сразу весь рефакторинг может принести достаточно проблем, чтобы вы бросили это гиблое дело на полпути до конца.

Будьте хирургом, который не боится отрезать больную часть, чтобы освободить место для здоровой. Такая позиция заразна и вскоре остальная команда присоединится к вашей инициативе по оздоровлению проекта. Список «гигиенических» процедур, которые ваша команда считает необходимыми, тоже хорошая практика в долгосрочной

перспективе. Убедите руководство в том, что хотя эти процедуры и не дают сразу видимых результатов, они приведут к снижению расходов и облегчат будущие релизы. Никогда не прекращайте поддерживать код в «здоровом» состоянии.

Автор оригинала - Mike Lewis

Не забывайте о "Hello, world"

(В оригинале - Learn to Say "Hello, World")

Paul Lee, имя пользователя leep, более известен как Hopry, имел репутацию эксперта в программировании. Мне требовалась помощь. Я подошел к его рабочему месту и спросил, не мог бы он взглянуть на мой код?

«Конечно», - сказал Hopry, отодвигая стул. Я постарался не зацепить пирамиду пустых банок из под колы, стоящую позади него. «Какой код?»

«В функции в файле», - ответил я.

«Что ж, давай посмотрим». Hopry отодвинул в сторону копию K&R и придвинул к себе клавиатуру.

Так, а где же IDE? Похоже, что у Hopry IDE не был запущен вообще, а был только какой-то редактор, которого я совсем не знал. Несколько нажатий клавиш – и файл открыт. На нужной функции (а файл был очень большим). Он пролистал функцию до того места, о котором я говорил.

«А что произойдет, если `x` будет отрицательным?» - спросил я. «Похоже, что так не должно быть».

Я потратил все утро на поиск способа сделать `x` отрицательным, но большая функция в большом файле была лишь частью очень большого проекта, и цикл перекомпиляции и перезапуска был слишком долгим для эффективной работы. Возможно, такой эксперт, каким был Hopry, мог бы просто дать мне ответ?

Hopry ответил, что он не уверен. К моему удивлению, он не стал ничего делать в K&R. Вместо этого, он скопировал часть кода себе в редактор, заменил в нем индентацию, сделал его функцией. После чего добавил еще одну функцию с бесконечным циклом, запрашивающую у пользователя входные данные, вызывающую нашу функцию и печатающую результат. Он сохранил это все в файл tryit.c. Все это я бы мог сделать сам, хотя и не так быстро. Однако следующий шаг был одновременно и простым, и полностью для меня новым.

```
$ cc tryit.c && ./a.out
```

Вау! Его программа, еще несколько назад бывшая лишь идеей, уже работала! Мы попробовали разные варианты входных данных и быстро подтвердили мои подозрения (хоть в чем-то я оказался прав). После чего он перепроверил отдельные

места в K&R. Я поблагодарил его и пошел к себе, снова постаравшись не зацепить стоящую позади него пирамиду пустых баночек.

Придя на свое место, я закрыл IDE. Я так долго работал над большим проектом с помощью больших инструментов, что я начал думать, что это именно то, что я и должен делать. Однако компьютеры способны решать и небольшие задачи. Я открыл текстовый редактор и начал печатать.

```
#include <stdio.h>

int main()
{
    printf("Hello, World\n");
    return 0;
}
```

Автор оригинала - Thomas Guest

Не игнорируйте ошибки.

(В оригинале - Don't Ignore that Error!)

> *Однажды вечером я шел по улице на встречу с друзьями в баре. Мы уже давно не встречались попить пива, поэтому я спешил на встречу. И в спешке я не смотрел под ноги. А зря. Тогда бы я вовремя заметил бордюрный камень, так некстати подвернувшийся мне под ногу.*
>
> *Споткнувшись, я, похоже, повредил ногу, но я же спешил на встречу! Я продолжил свой путь несмотря на то, что боль все усиливалась. Поначалу я ее почти не замечал, но по мере того, как боль нарастала, я начал подозревать, что с ногой что-то не так.*
>
> *Но ведь я же спешил на встречу с друзьями! И когда я добрался до бара, боль стала невыносимой. Я не получил никакого удовольствия из-за боли. А когда утром я пошел к врачу, оказалось, что у меня сложный перелом голени. Остановись я сразу, как только почувствовал боль, проблем было бы гораздо меньше – ходьба на сломанной ноге ни к чему хорошему не приводит. Возможно, это было худшим утром в моей жизни.*

Слишком много программистов пишет столь же неосмотрительно, как вел себя я той ночью.

Ошибка? Какая еще ошибка? Она наверняка несерьезная. Честно. Я могу не обращать на нее внимания. Однако это проигрышная стратегия. А если честно, это всего лишь лень (причем лень неконструктивная!). Какой бы ничтожной ни была на ваш взгляд ошибка в вашем коде, она обязательно должна быть исправлена или обработана. Всегда. Вы не выиграете время, если не сделаете этого. Вы лишь спрячете потенциальную проблему на будущее.

Мы оповещаем об ошибках во время выполнения различными способами:

- **Код возврата**. Он может применяться для оповещения о том, что «что-то пошло не так». Код возврата слишком просто проигнорировать. И вы ничего не заметите вообще. Более того, игнорирование кодов возврата некоторых стандартных функций стало уже традицией. Как часто вы проверяете код возврата функции printf()?
- **errno**. Это такая особенность C, глобальная переменная, хранящая код последней случившейся ошибки. Его легко проигнорировать, сложно использовать, к тому же его использование вызывает множество других проблем – представьте

многопоточную систему, в которой вызывается одна и та же функция. Некоторые платформы что-то делают, чтобы облегчить использование errno, другие не делают ничего.
- **Исключения**. Более структурированная языковая конструкция для сообщения об ошибках. К тому же, вы не можете их игнорировать. Или все же можете? Как насчет вот такого кода?

```
try {
    // ...do something...
}
catch (...) {} // ignore errors
```

Единственное оправдание этой ужасной конструкции – это то, что она явно показывает то, что вы делаете очень нехорошую вещь.

Если вы игнорируете ошибку в надежде, что ничего не случится, вы сильно рискуете. Как моя нога, пострадавшая гораздо сильнее, чем могла бы, если бы я тогда вовремя остановился. Если вы двигаетесь вперед без оглядки, вы можете столкнуться с очень серьезными проблемами. Решайте проблемы как можно раньше после их возникновения!

Игнорирование ошибок приводит к:

- **Ненадежному коду**. Коду, наполненному неуловимыми и сложноисправляемыми ошибками.
- **Небезопасному коду**. Хакеры часто используют игнорирование кодов ошибок, и очень часто делают это весьма успешно, «ломая» системы.
- **Плохо структурированному коду**. Если аккуратная обработка ошибок слишком скучна, скорее всего, это результат плохо продуманного интерфейса. Переделайте его так, чтобы обрабатывать ошибки было не так обременительно.

Кроме проверки всех потенциальных ошибок, нужно еще делать видимыми все потенциально возможные ошибочные состояния ваших интерфейсов. Не скрывайте их, расчитывая на то, что это будет работать.

- Почему мы не проверяем ошибки? Есть несколько общепринятых отговорок. С какими из них вы согласитесь? Обработка ошибок загромождает код, делая его нечитаемым и не позволяя легко отследить основной поток выполнения.
- Обработка ошибок требует дополнительных усилий, а у меня дедлайн надвигается.
- Я уверен, что эта функция никогда не вернет ошибку (printf работает всегда, malloc всегда выделит запрашиваемый объем памяти, а если нет – мы все равно ничего не сможем сделать, ведь это значит, что не работает вся система)

- Это несерьезная программа, нет смысла ее писать с промышленным качеством.

Автор оригинала - Pete Goodliffe

Не надейтесь на магию

(В оригинале - Don't Rely on "Magic Happens Here")

Если посмотреть на любую активность, процесс или дисциплину с достаточного расстояния, то все будет выглядеть просто. Менеджеры без опыта программирования считают программирование простой задачей, а программисты без опыта руководства думают то же самое о работе менеджеров.

Самая сложная часть процесса программирования – мышление – одновременно и самая незаметная часть, а также наименее ценимая непосвященными. В течении десятилетий было много попыток устранить из процесса необходимость наличия мышления. Одна из самых ранних и наиболее запомнившихся попыток – это усилия Грейс Хоппер по созданию более понятного языка программирования, что по мнению некоторых должно было вообще снять потребность в специальности «Программист». Результат (язык COBOL) внес существенный вклад в привлечение в отрасль новых программистов в последующие десятилетия.

Настойчивые попытки упростить разработку ПО так, чтобы вообще сделать программистов ненужными, для тех программистов, кто понимает, что из себя представляет процесс на самом деле, кажутся весьма наивными. Однако ментальный процесс, ведущий к такой ошибке, является частью человеческой натуры, и программисты столь же склонны к нему, как и все остальные.

В любом проекте есть много вещей, которые остаются за кадром для отдельных программистов: формирование требований, подтверждение бюджета, установка и настройка сервера сборки, процесс тестирования и сдачи в эксплуатацию, миграция бизнеса со старых процессов на новые и прочее.

И когда кто-то не вовлечен в процесс, то он склонен неосознанно предполагать, что этот процесс – простой и делается «чудесным образом». Пока магия работает, все идет хорошо. Но когда (причем именно «когда», а не «если») магия вдруг перестает работать, начинаются серьезные проблемы.

Я знаю проекты, терявшие недели из-за того, что никто не понимал того, насколько проект зависим от «правильной» версии используемой библиотеки. И когда начались проблемы, члены команды искали причины во всех остальных местах, пока наконец-то кто-то не догадался, что причина в неправильной версии библиотеки.

Другой пример – один из отделов разработки работал очень хорошо: проекты выполнялись вовремя, никаких ночных сессий экстренной отладки, никаких аварийных заплаток. Все шло настолько хорошо, что руководство решило, что это «хорошо» происходит само собой, и менеджер проекта им не нужен. И через полгода в этом отделе все стало также, как и во всей остальной компании – задержки, ошибки, заплатки...

Вам не нужно понимать всю магию, стоящую за вашим проектом, но хуже от того, что вы разберетесь с ее частью, точно не станет. По крайней мере цените тех, кто за эту магию ответственнен.

Что еще более важно, убедитесь, что магия снова может быть запущена, когда она вдруг перестает работать.

Автор оригинала - AlanGriffiths

Не повторяйтесь

(В оригинале - Don't Repeat Yourself)

Из всех принципов программирования «Не повторяйтесь» возможно один из самых фундаментальных. Принцип был сформулирован Энди Хантом и Дэйвом Томасом в книге *«The Pragmatic Programmer»* и лег в основу множества известных программистских практик и паттернов проектирования. Разработчик, умеющий находить повторения и понимающий как их избежать при помощи соответствующих практик или абстракций, может писать гораздо более аккуратный код по сравнению с тем, кто постоянно загрязняет приложение ненужными повторами.

Повторение – это потери

Каждая строчка кода, попавшая в производство, должна сопровождаться, и является возможным источником ошибок в будущем. Повторение раздувает код, давая больше возможностей для ошибок и внося дополнительное усложнение в систему. Раздутая система также затрудняет разработчикам понимание работы всей системы. К тому же становится сложно определить, что изменения в одной части достаточны, и не нужно их внести еще раз где-нибудь еще из-за повторения. «Не повторяйтесь» требует, чтобы правило «каждая часть знаний должна иметь единственное, недвусмысленное, официальное представление в системе» всегда соблюдалось.

Повторение процесса требует автоматизации

Множество процессов в разработке ПО повторяемы и легко автоматизируемы. Принцип «Не повторяйтесь» применяется здесь так же успешно, как и в исходном коде. Ручное тестирование медленно, ненадежно и сложно повторяемо, поэтому автоматизация тестирования должна применяться везде, где это возможно. Интеграция ПО может быть времязатратной и ненадежной, если делается вручную, поэтому процедура сборки должна выполняться как можно чаще, идеально при каждом помещении изменений. Там, где существует неприятный ручной процесс, который можно автоматизировать, то он должен быть автоматизирован. Цель – обеспечить, чтобы существовал единственный способ выполнить задачу, и этот способ должен быть как можно проще и легче.

Повторение логики требует абстракции

Повторение логики может принимать множество форм. Копи-паст, *if-then*, *switch-case* – вот самые простые способы определить проблемное место. Множество паттернов проектирования предназначены для уменьшения повторов в логике. Если объект требует выполнения нескольких действий перед использованием, это может быть решено при помощи Abstract Factory или Factory Method. Если объект имеет множетсво вариаций поведения, то это может быть реализовано паттерном Strategy вместо большой if-then структуры. Само описание паттернов проектирования – попытка снизить повторяемость усилий по решению часто встречающихся задач. Еще «Не повторяйтесь» может применяться к структурам данных (схемы баз данных), давая в результате нормализированную форму.

Основной принцип

Другие принципы разработки также основаны на «Не повторяйтесь». Например, принцип «Один и только один», применяемый только к функциональному поведению кода, может быть представлен как частный случай «Не повторяйтесь». Принцип «Открыто/Закрыто», говорящий, что «единица кода должна быть открыта для расширений, но закрыта для изменений», на практике работает лишь при условии следования принципу «Не повторяйтесь». Также и хорошо известный принцип «Единственность ответственности» требующий, чтобы класс имел «лишь одну причину для изменений», основывается на «Не повторяйтесь».

Будучи применимым к структуре, логике, процессу и функциональности, принцип «Не повторяйтесь» задает фундаментальное правило для разработчиков и ведет к созданию более простых, более сопровождаемых и более качественных приложений. И хотя существуют сценарии, в которых повторение может быть необходимым (например, требования к производительности), они должны применяться лишь там, где они действительно необходимы для решения реальных, а не воображаемых проблем.

Автор оригинала - Steve Smith

Не работайте сверхурочно

(В оригинале - Hard Work Does not Pay Off)

Если вы программист и много работаете, это не окупается. Вы и некоторые из ваших коллег можете верить в то, что сверхурочная работа на проекте приносит проекту пользу. Однако на самом деле работая меньше, вы можете получить лучшие результаты. Иногда во много раз лучшие. Если вы стараетесь быть «продуктивными» более чем 30 часов в неделю, скорее всего, вы слишком усердны. Вам нужно подумать о том, чтобы снизить нагрузку для того, чтобы стать более эффективным и получать больше результатов.

Это утверждение может казаться странным и противоречивым, однако оно напрямую следует из того факта, что разработка ПО требует постоянного процесса обучения и совершенствования. По мере работы над проектом вы будете понимать все больше проблем предметной области и находить более эффективные пути достижения целей. Чтобы избежать потерь времени и усилий, вам нужно постоянно отслеживать эффект от того, что именно вы делаете, изменяя свои действия при необходимости.

Профессиональное программирование меньше всего похоже на многокилометровый забег, где цель – в конце проложенной дороги. Большинство проектов больше похоже на спортивное ориентирование. В темноте. С эскизом вместо карты. Если вы при этом выберите одно направление и помчитесь туда сломя голову так быстро, как только можете, возможно, вы и впечатлите кого-нибудь, но вряд ли придете к финишу. Вам нужно идти с разумной скоростью, постоянно отслеживая путь, контролируя свое местоположение и внося необходимые поправки по мере продвижения вперед.

Кроме этого, вам постоянно необходимо совершенствоваться в различных программистских техниках. Вам необходимо читать книги, посещать конференции, общаться с другими профессионалами, экспериментировать с разными техниками и изучать эффективные инструменты, облегчающие вашу работу. Как профессиональному программисту, вам нужно постоянно оставаться на нужном уровне экспертности в своем деле – точно также, как если бы вы были нейрохирургом или пилотом реактивного самолета. Вам нужно тратить часть ваших вечеров, выходных и каникул на обучение, и поэтому вы никак не можете тратить ваши вечера и выходные на сверхурочную работу на проекте. Вы же не будете ожидать сверхурочной работы по 60 часов в неделю от нейрохирурга или пилота? Ведь им нужно постоянно самосовершенствоваться!

Сконцентрируйтесь на проекте, старайтесь делать работу как можно лучше, находите эффективные решения, повышайте свой уровень, отслеживайте обратную связь от ваших действий и адаптируйте свои действия в зависимости от результата. Не дискредитируйте себя и профессию, работая с упорством и эффективностью белки в колесе. Будучи профессиональным программистом, вы должны знать, что работа по 60 часов в неделю – это не то, чем следует заниматься. Действуйте, как профессионал – подготовка, действие, наблюдение, анализ, изменение.

Автор оригинала - Olve Maudal

Не трогай это!

(В оригинале - Don't Touch that Code!)

Такое наверняка случалось с каждым. Ваш код был залит на демо-сервер для тестирования, и руководитель верификации сообщает о найденной проблеме. Ваша первая реакция при этом: «О, я знаю, что не так, давайте я быстро это поправлю».

Проблема здесь в том, что вы думаете, что у вас как у разработчика должен быть доступ к демо-серверу.

В большинстве web-ориентированых разработок архитектура представляет собой что-то вроде:

- Разработку и юнит-тестирование на локальной рабочей станции
- Сервер разработки, где осуществляется интеграционное тестирование
- Демо-сервер, где верификаторы и заказчики осуществляют тестирование и проверяют критерии приемки
- Производственный сервер

Да, есть и другие сервера, вроде системы контроля версий и трекинга ошибок, но идею вы поняли. По этой модели разработчик, даже если он ведущий, не должен иметь доступа никуда кроме сервера разработки. Большая часть разработки осуществляется на локальном компьютере с любимым IDE, виртуальными машинами и другими магическими штуками, повышающими удачу.

После помещения в систему контроля версий код должен попасть на сервер разработки, где он будет протестирован на то, что вся система все еще работает. После этого разработчики могут лишь наблюдать за процессом.

Код отправляется на демо-сервер, где с ним начинает работать верификация. Точно также, как разработка не должна иметь доступа никуда, кроме сервера разработки, верификация не должна ничего трогать на сервере разработки. Не надо просить «проверить одну маленькую функциональность» непосредственно на сервере разработки. Вместо этого используйте нормальный цикл сборки и тестирования на демо-сервере. Разработка идет своим чередом, и текущее состояние может быть нежелательным для посторонних глаз. Менеджер, ответственный за релиз – единственный, кто может иметь доступ к обоим серверам.

И ни при каких обстоятельствах разработка не должна иметь доступа к производственному серверу! Если в поле возникла проблема, то техподдержка должна ее решить или же официально запросить решение у разработки. После того, как

решение будет помещено в систему контроля версий, техподдержка заберет исправление оттуда. Самые ужасные катастрофы, которые я наблюдал, происходили из-за того, что кто-то нарушил это право (надеюсь, никто не догадался, что этим кем-то был я сам). Если что-то сломалось, то производственный сервер – не место для исправлений!

Автор оригинала - Cal Evans

Непрерывное обучение

(В оригинале - Continuous Learning)

Мы живем в интересное время. Как только разработка ПО распространилась по всему земному шару, многим стало понятно, что в мире полно людей, способных делать их работу. И необходимо все время обучаться, чтобы представлять ценность на рынке. Иначе вы постепенно превратитесь в динозавра, «прилипшего» к своей работе, пока в один прекрасный день ваши услуги больше не будут нужны или же вашу работу переложат на более дешевый outsource.

И что же с этим делать? Некоторые работодатели достаточно щедры, чтобы оплачивать обучение сотрудников, другие могут не иметь достаточно ресурсов для этого. Вам потребуется взять ответственность за свое обучение в свои руки.

Вот список возможных для образования способов. Часть из них полностью бесплатно при наличии интернета.

- Читайте книги, журналы, блоги, твиттер и различные сайты. Если захотите «копнуть глубже», то подпишитесь на рассылку.
- Если захотите реально погрузиться в технологию – напишите какой-нибудь код.
- Старайтесь найти ментора, поскольку если вам не на кого равняться, это может сильно замедлить ваше обучение. Наиболее эффективно учиться у кого-то, кто имеет больше опыта или в чем-то лучше вас. Если не найдете ментора, все равно двигайтесь вперед сами.
- Не игнорируйте виртуальных менторов. Найдите в интернете авторов и программистов, на кого бы вы хотели равняться, и читайте все, что они напишут.
- Изучите фреймворки и библиотеки, используемые вами для работы. Если вы знаете, как оно работает, вы сможете это использовать гораздо эффективнее. Если вы имеете дело с open source, то считайте, что вам повезло – берите отладчик и шаг за шагом исследуйте, что там происходит внутри. Вы столкнетесь с кодом, написанным и проверенным очень способными людьми.
- Когда вы сделали что-то не так, исправляете ошибку или сталкиваетесь с проблемой, старайтесь всегда выяснить, что именно произошло. Очень вероятно, что такое уже случалось, и кто-то уже опубликовал решение. Надо только погуглить.
- Лучший способ чему-нибудь научиться – это научить кого-нибудь еще. Когда вас будет слушать много людей, а потом задавать вам вопросы, у вас будет отличная мотивация это выучить очень хорошо.
- Присоединитесь к сообществу (или откройте свое), где изучается язык, технология

или предмет, интересный для вас.
- Участвуйте в конференциях. Если нет возможности посещать их вживую, то многие из них выкладывают часть материалов онлайн.
- Долгая дорога на работу? Слушайте подкасты!
- Запускали когда-нибудь статический анализатор кода? Или хотя бы обращали внимание на warning-и в вашем IDE? Разберитесь, что они означают и почему появляются.
- Изучайте по новому языку программирования в год. Или хотя бы по новой технологии или инструменту. Это даст вам новые идеи, полезные в вашей текущей работе.
- Не обязательно изучать лишь технологии. Углубитесь в предметную область, с которой вы работаете, чтобы лучше понимать требования и находить решения проблем. Изучение того, как повысить свою производительность – еще одна очень полезная вещь, которую не стоит игнорировать.

Было бы замечательно, если бы люди имели такую возможность, как у Нео в «Матрице» - загружать необходимые знания непосредственно в мозг. Но увы, для этого потребуется время. Не стоит, конечно, тратить на обучение все свое свободное время, но стоит делать это регулярно. Немного времени раз в неделю – сильно лучше, чем вообще ничего. Оставьте себе и другую жизнь, кроме работы.

Технологии меняются быстро. Не останьтесь позади!

Автор оригинала - Clint Shank

Нет ничего более постоянного, чем временное

(В оригинале - The Longevity of Interim Solutions)

Почему мы создаем временные решения?

Обычно возникает какая-то срочная проблема. Она может быть внутренней – нехватка какого-либо инструмента для выполнения работы. Она может быть внешней, видимой конечным пользователям.

В большинстве систем вы можете найти пример того, что какой-то элемент слегка отличается от остальной системы, выглядит как черновик, который должен быть переписан в будущем, не соответствует стандартам, по которым выполнен весь продукт, и т.д. Вы наверняка услышите возмущения разработчиков по этому поводу. Причины такого могут быть различны, однако основная, из-за которой временные решения столь живучи, это то, что эти решения практичны.

Однако, временные решения обладают большой инертностью и устойчивостью. Они уже есть, они работают и почти всех устраивают, нет необходимости что-то менять прямо сейчас. Когда принимается решение о том, что сделать на следующем шаге, чтобы получить максимальную отдачу, наверняка найдется множество более важных вещей, чем замена временного решения на нормальное. Почему? Потому что оно уже есть, оно работает, оно всех устраивает. Единственная проблема – оно не соответствует тем или иным стандартам и требованиям, однако за исключением некоторых ниш этим обычно пренебрегают.

И таким образом временные решения остаются навсегда.

И если с временным решением возникнет какая-нибудь проблема, вряд ли стоит ожидать, что в этот момент оно будет полностью переписано «как надо». Скорее всего, будет сделано временное решение проблемы временного решения, которое опять-таки всех устроит.

Есть ли здесь проблема?

Ответ зависит от конкретного проекта и вашего отношения к стандартам кодирования. Если в системе слишком много временных решений, ее энтропия (или внутренняя сложность) возрастает, а сопровождаемость снижается. Однако, это не тот вопрос, который следует задавать в первую очередь. Мы сейчас говорим о решениях.

Возможно, это не является вашим предпочтительным решением (вряд ли это решение предпочтительно для кого-либо), однако это – решение, и мотивация для его переделки будет низкой.

Что же мы можем сделать, если столкнемся с такой проблемой?

1. Избегать временных решений с самого начала
2. Поменять силы, влияющие на решения менеджера проекта
3. Оставить все так, как есть

Давайте рассмотрим все три варианта подробнее.

1. Избежать временного решения получится не всегда. Есть реальная проблема, которую надо решить, а стандарты слишком жесткие и сделать этого не позволяют. Вы можете потратить какое-то количество энергии на попытку поменять стандарты. Достойно уважения, хотя и утомительно. И неэффективно с точки зрения вашего проекта и проблемы.
2. Упомянутые силы основаны на культуре проекта, что затруднит изменения. Возможно, это и получится, особенно если проект маленький и вы можете сами все разгрести без чьего-либо одобрения. Или же если проект уже настолько погряз во временных решениях, что это уже видно и снаружи, и на его расчистку официально выделяется время.
3. Если же вы не сделаете ни первое, ни второе, то существующее положение вещей таковым и останется.

Вам придется решать множество проблем, часть решений будут временными, большинство из них будут практичными. Наилучший способ справиться со временными решениями – это делать их заведомо избыточными, чтобы потом предоставить более элегантное решение. Возможно, у вас хватит спокойствия принять то, что вы не можете поменять, мужества менять то, что вы можете, и мудрости отличить одно от другого.

Автор оригинала - Klaus Marquardt

О комментариях

(В оригинале - A Comment on Comments)

На самых первых курсах программирования в колледже преподаватель однажды выдал по два бланка «Программа на Basic», написал на доске «Напишите программу, подсчитывающую среднее значение 10 игр в боулинг» и вышел из класса. Было ли это сложно? Я уже не помню деталей, но скорее всего, там был цикл и не более 15 строчек кода. Бланки (да-да, нам приходилось сначала писать программы на бумажных бланках перед тем как вводить их в компьютеры) позволяли написать до 70 строк кода. Мне было непонятно, зачем был нужен второй. Поскольку с аккуратностью у меня были проблемы, я переписал программу на второй бланк максимально аккуратно, надеясь получить дополнительные баллы.

Однако когда я получил проверенный бланк на следующем занятии, я с удивлением обнаружил там весьма мало баллов. Поперек тщательно написанной мной программы красовалась небрежная надпись «Без комментариев?».

То, что и я, и преподаватель знали, что должна делать программа, было недостаточно. Частью обучения была попытка показать, что код должен быть понятным и следующему программисту, пришедшему после меня. И я никогда не забуду этот урок.

Комментарии – это такой же необходимый элемент в программировании, как например условия или циклы. Многие современные языки программирования даже имеют инструмент, позволяющий из специально отформатированных комментариев сгенерировать API документацию. Это очень неплохо, но все же недостаточно. Внутри самого кода должны быть комментарии, объясняющие, что этот код делает. Программирование в стиле «То, что было сложно написать, должно быть не менее сложно прочесть» навредит всем – вашему заказчику, вашему работодателю, коллегам в команде и вам самому в ближайшем будущем.

С другой стороны, не стоит заходить слишком далеко, комментируя все подряд. Комментарии должны прояснять ваш код, а не загромождать его. Лишь немного «сбрызните» код комментариями, объясняющими сложные места. «Заголовочные» комментарии должны предоставлять любому программисту достаточно информации о том, как использовать ваш код, вообще в него не заглядывая. А «внутренние» комментарии должны помочь следующему за вами разработчику сопровождать и изменять ваш код.

На одной своей работе я был несогласен с архитектурным решением, принятым наверху. Испытывая раздражение, часто присущее молодым программистам, я вставил кусок письма с указанием использовать требуемое решение непосредственно в «шапку» файла. Оказалось, что менеджеры просматривали изменения перед каждым коммитом. Так я познакомился со значением термина *«действие, ограничивающее карьерный рост»*.

Автор оригинала - Cal Evans

О пользе изобретения велосипеда

(В оригинале - Reinvent the Wheel Often)

«Используйте уже написанное – глупо изобретать велосипед...»

Приходилось ли вам слышать что-то подобное? Конечно же! Каждый разработчик не один раз слышал такое. Хотя почему? Почему так плохо изобретать велосипед? Потому что, чаще всего, уже существующий код – это работающий код. Он уже прошел через контроль качества, тестирование и успешное использование. К тому же, усилия, затраченные на разработку аналога уже существующего кода, скорее всего, не окупятся. Так стоит ли изобретать велосипед?

Наверняка вы читали статьи и книги о паттернах проектирования или дизайне ПО. К сожалению, часто такие книги оказываются весьма бесполезными независимо от того, насколько классная информация в них предоставлена. Примерно такая же разница между «посмотреть фильм о мореплавании» и «выйти в море самому». И то же самое можно сказать и об использовании существующего кода по сравнению с проектированием и написанием кода с нуля, его тестированием, отладкой, исправлением и улучшением в течение всего жизненного цикла.

Изобретение велосипеда – это не только упражнение на написание кода. Это способ получить глубокие знания внутреннего устройства различных уже существующих компонент. Знаете ли вы, как работают менеджеры памяти? А виртуальные страницы? А сможете это реализовать самостоятельно? А двойные списки? А динамические массивы? ODBC клиенты? А получится ли у вас написать графический интерфейс пользователя, работающий так же, как и ваш любимый? А как насчет отличия мультизадачной и мультипотоковой систем? А правильно выбрать размещение базы данных – на диске или в памяти? Большинство разработчиков никогда самостоятельно не разрабатывало эти базовые вещи и поэтому не имеет глубоких знаний о том, как они все работают. Для них это все – лишь загадочные черные ящики, которые просто как-то работают. Понимание лишь того, что происходит на поверхности воды, недостаточно для избегания всех опасностей, скрывающихся в глубине. Непонимание глубины основ в разработке ПО послужит серьезным препятствием для написания чего-либо выдающегося.

Изобрести велосипед неправильно – более ценно, чем сразу узнать о том, как это надо делать, прочитав книгу. В этом случае вы получите урок по принципу «попробовал – не получилось», что гораздо более эмоционально, чем просто чтение технической книги.

Изучение фактов из книг – хорошо и важно, но стать сильным программистом вам в гораздо большей степени поможет опыт, а не факты. Изобретение велосипеда для разработчика ПО – весьма важная часть в его опыте и образовании.

Автор оригинала - Jason P Sage

Обмен сообщений вместо разделяемой памяти

(В оригинале - Message Passing Leads to Better Scalability in Parallel Systems)

Программистам говорят с самого начала их обучения, что параллельные вычисления – это сложно, что только лучшие могут с этим совладать, и даже они все равно будут делать ошибки. При этом особое внимание уделяется потокам, семафорам, мониторингу и сложности обеспечения безопасного параллельного доступа к переменным.

Да, действительно, в этом много проблем и эти проблемы сложно решаются. Но что является источником всех проблем? Общая память. Практически все проблемы параллельного программирования вырастают из общей изменяемой памяти: гонки (race condition), блокирования (deadlocks) и прочее. Решение же очень простое: выбирайте что-то одно. Или параллелизм, или общую память.

Как вы понимаете, отказ от параллельного программирования – не вариант. Компьютеры обзаводятся все большим и большим количеством ядер, поэтому параллельное программирование становится все более важным. Мы не можем продолжать надеяться на повышение частот процессоров для повышения производительности. И только параллельные вычисления позволяют повышать производительность и далее. Конечно, можно отказаться от повышения производительности, но вряд ли это понравится пользователям.

Получается, что нам нужно отказаться от общей памяти.

Вместо использования потоков и общей памяти нам нужно перейти к модели процессов и передачи сообщений. Под процессом тут подразумевается защищенный независимый код, а не процесс операционной системы. Языки вроде Erlang показали, что процессы – очень удачный механизм для программирования многозадачных систем. У таких систем отсутствуют проблемы синхронизации, обычные для систем с потоками и разделяемой памятью. Более того, существует формальная модель коммуникации последовательных процессов (Communication Sequential Processes), использующаяся при разработке подобных многозадачных систем.

Мы можем пойти дальше и представить системы с потоками данных как способ вычислений. В системах с потоками данных не существует заданного потока вычислений. Вместо этого задается граф операторов, соединенных путями данных,

после чего данные поступают в систему. Процесс контролируется готовностью данных. И никаких проблем синхронизации.

Однако системные языки вроде C, C++, Java, Python представляются программистам как языки для разработки многопоточных приложений с распределенной памятью. Что же с этим делать? Ответ – использовать (а если таковых не будет – то создавать) библиотеки и фреймворки, предоставляющие модель процессов и передачи сообщений и избегающие использования общей изменяемой памяти.

Программирование с использованием сообщений вместо разделяемой памяти – наиболее успешный способ разработки параллельных систем, преобладающих сейчас компьютерной индустрии. Возможно, что в будущем потоки будут использовать для реализации процессов, как бы странно это сейчас не звучало.

Автор оригинала - Russel Winder

Одна голова - хорошо, а две - лучше

(В оригинале - Two Heads Are Often Better than One)

Программирование требует сосредоточенности, а сосредоточенность требует уединения. Таков один из стереотипов.

Подход «одинокого волка» потихоньку меняется на более коллективный, который позволяет для программистов повысить качество, продуктивность и удовлетворенность от работы. Согласно новому подходу, разработчикам приходится более тесно сотрудничать друг с другом и с не-разработчиками – бизнес-аналитиками, системными аналитиками, тестерами и пользователями.

Что это означает для программистов? Быть техническим экспертом теперь недостаточно, нужно еще быть эффективным в работе с другими.

Сотрудничество – это не задавание вопросов или сидение на митингах. Это, закатав рукава, сделать работу вместе с кем-то.

Я большой фанатик парного программирования. Вы можете назвать это «экстремальным сотрудничеством». Как разработчик, мои умения растут, когда я работаю в паре. Если я слабее своего партнера в предметной области или технологии, я обучаюсь с его помощью. Если я сильнее в чем-то, то я еще лучше разбираюсь в том, что знаю и чего не знаю, объясняя это партнеру. Мы оба в паре даем что-то друг другу и учимся друг у друга.

В паре мы создаем коллективный опыт, как в предметной области, так и в технической части, и это позволяет нам писать ПО более разумно и эффективно. Даже в случае сильного дисбаланса уровня внутри пары, более опытный все равно что-то получает взамен, скажем, новую комбинацию «горячих клавиш» или же опыт с новым софтом или библиотекой. Для менее опытного в такой паре – это замечательная возможность быстро «разогнаться».

Парное программирование популярно среди поклонников agile разработки, хотя и не ограничивается ими. Те, кто возражает против парного программирования, обычно задают вопрос «Почему я должен платить двум программистам за работу одного?». Мой ответ в этом случае «Не надо». Я утверждаю, что парное программирование повышает качество, понимание предметной области, владение техникой (IDE и другие программы), а также снижает последствия «риска лотереи» (если один из ваших ведущих разработчиков выиграет в лотерею и уволится на следующий день).

Но какова долгосрочная ценность изучения нового сочетания клавиш? Как измеряется повышение качества результирующего продукта в результате введения парного программирования? Как оценить влияние партнера, не дающего вам довести до конца решение сложной задачи? Одно из исследований показывает повышение эффективности и скорости на 40% (J. T. Nosek, "The Case for Collaborative Programming," Communications of the ACM, Март 1998 года). А какова ценность снижения «риска лотереи»? Большинство подобных показателей достаточно сложно поддаются измерению.

Кто с кем должны образовать пары? Если вы новичок в команде, то важно найти кого-то, кто является экспертом. Точно также важно, чтобы он обладал навыками коуча. Если вы не сильны в предметной области, то образуйте пару с экспертом в предметной области.

Если вы еще не убеждены, то проведите эксперимент. Объединитесь с коллегами. Образуйте пары по интересным, сложным проблемам. Посмотрите, как оно пойдет. Попробуйте повторить эксперимент несколько раз.

Автор оригинала - Adrian Wible

Основы bug tracking-а

(В оригинале - How to Use a Bug Tracker)

Ошибки, *баги*, *дефекты* или даже *побочные эффекты дизайна* – называйте их как угодно, но это не поможет вам от них избавиться. А вот знание того, что такое хороший отчет об ошибке и на что в нем обращать внимание, очень даже может помочь.

Хороший отчет об ошибке должен содержать три вещи:

- как можно более точное описание того, как ошибку воспроизвести, и как часто эта ошибка воспроизводится;
- что должно было произойти, или как минимум ваше ожидание того, что должно было произойти;
- что произошло вместо этого, или по крайней мере как можно больше информации об этом.

Объем и качество информации, содержащейся в отчете об ошибке, так же хорошо говорит о качествах того, кто этот отчет создал, как и о самой ошибке. Злобное «Эта функция – полное дерьмо» сообщит разработчику лишь о том, что у нашедшего был тяжелый день, и ничего более. Описание ошибки, содержащее достаточно информации, чтобы ее легко воспроизвести, вызывает уважение, даже если из-за него приходится отложить релиз.

Ошибки – как диалог, со всей историей, видимой всем участникам. Не обвиняйте никого, и не отрицайте существование ошибки. Вместо этого попросите предоставить больше информации, указав по возможности, чего именно вам не хватает.

Изменение статуса ошибки, например, с *Open* на *Close* – публичное заявление того, что вы думаете по поводу ошибки. Найдите время обосновать ваше решение, это поможет сохранить время на пререкания с раздраженным менеджером или заказчиком. Изменение приоритета ошибки – такое же публичное заявление, а то, что вам ошибка кажется несущественной, не остановит от прекращения использования вашего продукта того, для кого она стала последней каплей.

Не перегружайте значения существующих полей отчета об ошибках. Добавление тега VITAL может и облегчит вам сортировку важного для вас отчета, но рано или поздно этот тег может быть скопирован кем-то еще, и когда-нибудь кто-нибудь случайно допустит в нем опечатку, или же для какого-нибудь нового отчета его придется удалять. Вместо этого используйте новое поле, и обязательно документируйте его использование.

Убедитесь, что все знают, как находить ошибки, за которые отвечает команда. Обычно это делается через сформированный запрос с понятным названием. Проверьте, что все используют один и тот же запрос, и не вносите в него изменения без информирования всей команды.

И главное – помните, что попытка стандартизировать работу с ошибками примерно столь же эффективна, как и измерение работы программиста числом строк написанного кода.

Автор оригинала - Matt Doar

Осознанная практика

(В оригинале - Do Lots of Deliberate Practice)

Осознанная практика – это не просто выполнение работы. Если вы спросите себя «Для чего я решаю эту задачу?» и ответом будет «Чтобы ее решить», то вы не заниметесь осознанной практикой.

Вы практикуетесь для того, чтобы улучшить свою способность к решению задач. Речь идет о мастерстве и технике. Практика означает повторение. Это означает выполнять задачу с целью повышения мастерства в одном или нескольких аспектах зачачи. Это означает повторять уже повторенное. Снова и снова. Пока не достигнете желаемого уровня мастерства. Вы практикуетесь, чтобы усвоить задачу, а не чтобы ее просто решить.

Главная цель оплачиваемой разработки – завершить продукт, в то время как главная цель совершенствования – повысить свою производительность. Это не одно и то же. Спросите себя, сколько времени вы тратите на разработку продукта для кого-то, а сколько – на развитие себя?

Сколько необходимо практиковаться, чтобы достичь мастерства? Peter Norvig как-то написал, что «Возможно, 10 тысяч часов – то самое волшебное число». Mary Poppendieck в *«Leading Lean Software Development»* заметила, что «Хорошему исполнителю потребуется минимум 10 тысяч часов практики, чтобы стать экспертом».

Экспертность не приходит вся сразу по окончании десятитысячного часа, а накапливается все это время постепенно. И все же 10000 часов – это много. Это по 20 часов в неделю в течении 10 лет. Вам может показаться, что просто не каждый может стать экспертом. Однако исследования последних десятилетий показывают, что основной фактор в становлении экспертности – именно осознанная практика, а не прирожденная способность.

- Mary: «Практически все исследователи экспертности сходятся в том, что врожденная способность лишь начало. Необходимо иметь самый базовый уровень, чтобы начать. После чего успеха достигают те, кто больше других работает над этим.»

Не имеет особого смысла практиковаться в вещах, в которых вы уже эксперт. Осознанная практика – практика в том, где вы еще не столь сильны.

- Peter: «Ключевой фактор в развитии экспертности – практика. Не просто делать что-то снова и снова, но ставить себе задачи чуть выше вашего уровня, пытаться

их решать, анализируя свою эффективность и исправляя ошибки».
- Mary: «Осознанная практика не значит делать то, в чем вы сильны, это значит бросать себе вызов, делая то, в чем вы слабы. Это не обязательно удовольствие».

Осознанная практика – это обучение. Обучение, которое вас меняет. Обучение, которое меняет ваше поведение. Удачи!

Автор оригинала - Jon Jagger

Осторожнее с повторным использованием!

(В оригинале - Beware the Share)

Это был мой первый реальный проект. Я только что закончил университет и старался хорошо показать себя, задерживаясь по вечерам и разбираясь в написанном коде. Во время реализации своей первой функциональности я особо заботился, чтобы использовать все, чему я научился – комментирование, лог, помещение общего кода в библиотеки там, где это возможно. Однако ревью кода стало для меня грубым возвращением к реальности – повторное использование кода категорически не было одобрено.

Как же это было возможно? Ведь везде говорили, что повторное использование – одна из основных вещей профессиональной разработки! Все статьи, которые я читал, все книги, все профессионалы, обучавшие меня – все говорили именно так! Где же была ошибка?

Оказалось, я пропустил одну критическую вещь.

Контекст.

То, что в двух никак не связанных частях системы оказалась одинаковая логика, значило гораздо меньше, чем я думал. До того как я вынес эту общую логику во внешнюю библиотеку, эти две части никак друг от друга не зависели. Каждая часть могла меняться независимо от другой. В каждой части логика могла меняться при необходимости в зависимости от требования бизнес окружения. Эти строки одинакового кода не были ничем иным как простым совпадением. До тех пор, пока не пришел я.

Общая внешняя библиотека, которую я создал, связала обе части вместе. В результате уже нельзя было изменить одну часть без синхронизации изменения со второй частью. Если ранее стоимость сопровождения этих двух независимых функциональностей была незначительна, теперь, после появления общей библиотеки, необходимость тестирования значительно возросла.

Одновременно с уменьшением общего количества кода в системе я увеличил количество зависимостей в ней. Контекст этих зависимостей весьма важен – находись эти две функциональности рядом, такая зависимость могла бы быть оправдана.

Однако когда функциональности вообще никак не связаны друг с другом, такая тесная зависимость становится проблемой, несмотря на то, что внешне код выглядит вполне нормально.

Эти ошибки коварны тем, что изначально выглядят как хорошие идеи и оптимизации. Будучи примененными в правильном контексте, эти техники могут принести пользу. В неправильном контексте их применение повысит лишь расходы, но не принесет пользы. И теперь, имея дело с написанным кодом без подробной информации о том, в каком контексте используются его отдельные части, я гораздо более осторожно отношусь к повторному использованию.

Будьте осторожны с повторным использованием. Сначала проанализируйте контекст, и только потом реализуйте.

Автор оригинала - Udi Dahan

Осторожно выбирайте внешние модули

(В оригинале - Choose Your Tools with Care)

Современные приложения редко пишутся с нуля. Они собираются из уже существующих «кирпичиков» - компонент, библиотек и фреймворков. Делается это из-за следующих причин:

- Приложения все сильнее растут в размерах, сложности и функциональности, а времени на разработку выделяется все меньше. Такой подход позволяет снизить время разработки и сконцентрироваться на написании бизнес-ориентированной части кода, а не на написание инфраструктурной части.
- Распространенные компоненты скорее всего содержат меньше ошибок, чем разработанные самостоятельно с нуля.
- В интернете есть много качественных программ с открытым кодом, доступных бесплатно, что также снижает расходы на разработку и повышает шансы найти разработчиков с требуемыми знаниями.
- Разработка и сопровождение ПО – достаточно ресурсозатратное дело, поэтому может быть дешевле купить готовое решение.

Однако, выбрать правильный набор инструментов для построения вашего приложения может оказаться нетривиальной задачей. Делая выбор, имейте в виду следующее:

- Различные инструменты могут иметь различные требования к контексту использования – инфраструктуре, модели управления, модели данных, протоколам коммуникации, что может привести к несоответствию между разными инструментами. Такое несоответствие приводит к различным хакам и обходным путям, что вносит излишнее усложнение в код.
- Разные компоненты могут иметь различный жизненный цикл, и обновление одного из них может оказаться крайне сложной задачей из-за того, что новая функциональность или даже исправление ошибки может оказаться несовместимым с остальными компонентами. И чем больше компонент используется, тем сложнее.
- Некоторые инструменты требуют значительных объемов конфигурации, часто в виде XML файлов, и эти файлы конфигурации могут выйти из под контроля очень быстро. В результате приложение будет выглядеть написанным на XML с

небольшим количеством вставок кода на другом языке. Сложность конфигурирования может сделать приложение сложным для сопровождения и модификации.

- При сильной зависимости от решений какого-либо производителя проблемы могут начаться в ситуации, когда производитель что-нибудь изменит: возможность поддержки, производительность, цену или что-нибудь еще.
- Если вы решите использовать бесплатное ПО, может оказаться, что оно совсем не бесплатное. Возможно, придется оплачивать коммерческую поддержку или коммерческую лицензию. Например, может оказаться неприемлемым использование ПО под бесплатной лицензией GNU, потому что результат нужно будет распространять под такой же лицензией GNU, с открытым исходным кодом.

Моя собственная стратегия избегания этих проблем – начинать с наименьшим возможным количеством используемых инструментов. Обычно вначале фокус на инструментах, реализующих низкоуровневое программирование, например, какой-нибудь прослойки над обычными сокетами операционной системы. А потом – добавлять по мере необходимости. Кроме этого, я стараюсь изолировать внешнее ПО от своей реализации, применяя для этого интерфейсы и слои. В результате я могу легко заменить внешний инструмент при необходимости. Положительный эффект такого подхода – завершенный продукт оказывается меньшим и с меньшим количеством используемых составляющих, чем ожидалось при проектировании.

Автор оригинала - Giovanni Asproni

Отойдите от клавиатуры

(В оригинале - Put the Mouse Down and Step Away from the Keyboard)

Вот уже несколько часов вы напряженно ищете решение весьма неприятной проблемы, но решение никак не находится. В какой-то момент вы все бросаете и идете размять ноги и выпить глоток кофе, а по пути назад вам вдруг приходит понимание того, как эту проблему можно решить.

Звучит знакомо, не так ли? Интересно, как это работает? А дело в том, что пока вы напряженно работаете, логическая часть вашего мозга активна, а интуитивная – нет. И это может продолжаться до тех пор, пока вы не дадите логической части отдохнуть.

Вот пример из жизни. Я подчищал старый код и наткнулся на «интересную» функцию. Функция должна была проверять, что в строке содержится правильное время в формате hh:mm:ss xx, где hh – часы, mm – минуты, ss – секунды, а xx это AM или PM.

Функция использовала следующий код для конвертации двух символов (значение часов) в число с последующей проверкой того, что значение лежит в нужном диапазоне:

```
try {
    Integer.parseInt(time.substring(0, 2));
} catch (Exception x) {
    return false;
}

if (Integer.parseInt(time.substring(0, 2)) > 12) {
    return false;
}
```

Подобный код повторялся еще два раза – для проверки минут и секунд. А завершался проверкой на AM или PM.

```
if (!time.substring(9, 11).equals("AM") &
    !time.substring(9, 11).equals("PM")) {
    return false;
}
```

Если все проверки проходили, возвращалось true, если хотя бы одна не проходила, возвращалось false.

Если вам показалось, что приведенный код написан не лучшим образом, то не волнуйтесь, мне показалось то же самое. Я захотел переписать это, написав сначала несколько юнит-тестов, чтобы быть уверенным, что все работает.

После окончания рефакторинга я остался доволен. Новая версия была гораздо легче читаема, в два раза компактнее и лучше протестирована.

А когда я пришел на работу на следующее утро, в голове у меня вдруг возникла идея: «А почему бы не использовать регулярные выражения?» И после нескольких минут кодирования я получил работающую реализацию, уместившуюся в одну строчку:

```
public static boolean validateTime(String time) {
    return time.matches("(0[1-9]|1[0-2]):[0-5][0-9]:[0-5][0-9] ([AP]M)");
}
```

Мораль истории не в том, что я в конце концов заменил тридцать строк кода одной, а в том, что пока я не отошел от компьютера, я думал, что моя первая попытка была лучшим решением из всех возможных.

Так что когда в следующий раз вы столкнетесь с сложнорешаемой проблемой, помогите себе. Как только вы вникнете в проблему, сделайте что-то, что вовлечет интуитивную часть мозга – послушайте музыку, пройдитесь вокруг. Иногда лучшее, что вы можете сделать для решения проблемы – это отойти подальше от мышки и клавиатуры.

Автор оригинала - BurkHufnagel

Перегруженный журнал ошибок может лишить вас сна

(В оригинале - Verbose Logging Will Disturb Your Sleep)

У системы, запущенной в эксплуатацию, первым признаком наличия проблем является перегруженный лог. Вы понимаете, что я имею в виду – когда каждый клик мышью в нормальном процессе работы приводит к тому, что единственный лог системы записывается куча сообщений. Слишком детальное логирование столь же бесполезно, как и его отсутствие.

Если вы разрабатываете системы, подобные моим, то после того, как ваша работа заканчивается, начинается работа других людей. После запуска системы в эксплуатацию она, скорее всего, будет работать некоторое, иногда достаточно продолжительное время, если повезет. Как вы собираетесь узнавать о том, что что-то в системе не так, и что вы будете с этим делать?

Возможно, кто-то будет мониторить систему, возможно, вы это будете делать сами. В любом случае, логи скорее всего будут частью мониторинга. И если что-то поднимет вас посреди ночи, вы хотите быть уверены в том, что для этого была веская причина. Если моя система решила умереть, я хочу об этом знать. Но если это лишь небольшой сбой, я предпочту спокойно поспать до утра.

Для многих систем первым индикатором того, что что-то пошло не так, является запись сообщения в лог. Скорее всего, в лог ошибок. Сделайте так, чтобы с первого дня работы системы каждая запись в лог ошибок приводила к немедленному оповещению вас, даже посреди ночи. Если вы можете имитировать нагрузку во время тестирования, то чистый журнал ошибок – первый сигнал о том, что система достаточно устойчивая. Или же первое предупреждение для вас, если это не так.

Распределенные системы вносят в игру еще один уровень сложности. Вам нужно решить, что вы будете делать, если откажет удаленный компонент. Для распределенных систем такой отказ – это в порядке вещей. Учитывайте это, когда проектируете политику ведения лога ошибок.

В целом, лучший индикатор того, что с системой все хорошо – очень редкие низкоприоритетные события в логе. Я буду счастлив, если буду получать одно сообщение низкого приоритета на каждое значимое событие в приложении.

Перегруженный лог – это индикатор того, что систему будет сложно контролировать после запуска. Если вы не ожидаете того, что в журнале ошибок будет что-то появляться, то вам будет гораздо проще понять, что нужно делать, когда там что-то однажды появится.

Автор оригинала - Johannes Brodwall

Перед началом рефакторинга

(В оригинале - Before You Refactor)

В какой-то момент каждому программисту приходится делать рефакторинг уже написанного кода. Но перед тем, как начинать рефакторинг, подумайте над нижеприведенными пунктами – это сохранит вам много времени и избавит от множества проблем.

- *Лучше всего реструктуризацию начать с подробной ревизии существующего кода и тестовых сценариев для этого кода*. Это поможет вам выявить сильные и слабые стороны существующего решения, чтобы потом оставить удачные моменты и избежать проблемных мест. Все мы часто думаем, что сможем написать лучше, чем уже написано... до тех пор пока не получаем в итоге что-то, гораздо худшее, чем предыдущая реализация, потому что мы ничему не научились на опыте и ошибках существующей системы.
- *Избегайте намерения переписать все*. Лучше стараться повторно использовать как можно больше уже написанного кода. Неважно, каким бы ужасным он не был, он уже оттестирован, просмотрен и прочее. Выбросить старый код, особенно если система уже «в поле», означает выбросить месяцы или даже годы, потраченные на оттестированный и прошедший проверку «в поле» код, имеющий при этом какие-то исправления, о которых вы не знаете. Без информации об этих «подводных камнях» может получиться так, что ваш переписанный код будет содержать те же «мистические» ошибки, которые уже были исправлены в старом коде. Это приведет к серьезным потерям времени, усилий и знаний, собираемых долгое время до этого.
- *Несколько последовательных изменений лучше, чем одно большое*. Небольшие изменения позволяют легче отслеживать воздействие на систему, например, при помощи тестов. Ничего приятного в том, чтобы после исправления обнаружить сотню провалившихся тестов. Это может вызвать раздражение и как следствие, принятие не самого лучшего решения. Тогда как пара-тройка провалившихся тестов легко укажет на источник проблемы.
- *После каждой итерации важно убедиться, что все существующие тесты все еще проходят*. Добавьте новые тесты, если существующие недостаточно покрывают изменяемый вами кода. Не выбрасывайте старые тесты без

основательного анализа. На первый взгляд тесты могут казаться незначительными и неважными для вас, но может оказаться полезным копнуть глубже, чтобы понять, зачем же они были добавлены.

- *Личные предпочтения и эго не должны быть ведущим фактором*. Если что-то работает, зачем это исправлять? То, что стиль или структура не соответствует вашим персональным предпочтениям, недостаточно для начала реструктуризации. Мысль о том, что вы можете написать лучше, чем ваш предшественник, тоже не является достаточной причиной.
- *Новая технология также недостаточна для рефакторинга*. Одна из худших причин переписать код – это то, что существующий код находится «в стороне» от сегодняшних «прикольных» технологий, и мы уверены, что новый язык или фреймворк могут сделать то же самое гораздо более элегантно. Если только анализ не покажет, что новый язык или платформа действительно не приведут к значительному улучшению функциональности, сопровождаемости или производительности, обеспечив при этом финансовый выигрыш, то лучше рефакторинг не начинать.
- *Помните, что люди ошибаются*. Реструктуризация не гарантирует, что новый код получится лучше (или хотя бы таким же), чем предыдущая версия. Я участвовал в нескольких неудачных попытках реструктуризации. Это не было приятно, но это было по-человечески.

Автор оригинала - Rajith Attapattu

Пишите код так, как будто вы будете сопровождать его до конца жизни

(В оригинале - Write Code as If You Had to Support It for the Rest of Your Life)

Вы можете спросить 97 человек о том, что должен знать и делать программист, и услышать 97 различных ответов. Это пугающе много. Все советы могут быть хорошими, все принципы – разумными, а истории – захватывающими, но с чего же начать? И что еще более важно, как, начав, сохранить приверженность наилучшим практикам, сделав их неотъемлемой частью своего программистского опыта?

Я думаю, что ответ лежит в вашем отношении. Если вы не заботитесь о своих программистах, тестерах, менеджерах, продавцах и пользователях, то вы никогда не решите начать работать по методологии Test-Driven разработки или же писать ясные комментарии в своем коде. Мне кажется, есть простой способ все время настраивать себя на максимальное качество конечного продукта:

> *Пишите код так, как если бы вам требовалось его сопровождать до конца жизни.*

Вот так просто. Если вы примете это, то случится множество замечательных вещей. Если вы согласитесь с тем, что любой из программистов имеет право позвонить вам посреди ночи и спросить, что же делает вами написанная функция fooBar, то вы очень сильно повысите свой уровень в программировании. Вы захотите давать более понятные имена переменным и функциям. Вы будете избегать блоков кода длиной в сотни строк. Вы будете изучать и применять паттерны проектирования. Вы будете писать комментарии, тестировать свой код и непрерывно его улучшать. У вас просто не будет выбора, кроме как стать лучше, опытнее и эффективнее.

На самом деле код, написанный вами много лет назад, все еще оказывает влияние на вашу карьеру, нравится вам это или нет. Вы оставляете следы своего знания, отношения, упорства, профессионализма, обязательности и удовлетворенности с каждым методом, классом и модулем, который вы проектируете и реализуете. Люди будут формировать свое мнение о вас, основываясь на вашем коде, который им доведется увидеть. И если это мнение будет все время негативным, то карьера не будет продвигаться так, как бы вы этого хотели. Позаботьтесь о своей карьере, о своих клиентах и своих пользователях в каждой строчке своего кода – пишите код так, словно вам нужно будет сопровождать его до конца жизни.

Автор оригинала - Yuriy Zubarev

Пишите маленькие функции, используя примеры

(В оригинале - Write Small Functions Using Examples)

Мы все хотели бы писать правильный код, а также убеждаться в том, что он правильный. В обоих случаях нам может помочь размер функции. Только не тот размер, который измеряется количеством строк кода (хотя этот параметр тоже интересен), а размер математической функции, которую мы реализуем.

Например, в игре Го есть положение, называемое «атари», в котором камни игрока могут быть пойманы его противником. Камень с двумя или более смежными свободными местами – не в положении «атари». Подсчитать количество свободных мест у камня может быть сложно, но после того, как оно подсчитано, определить «атари» очень легко. Первая версия функции может выглядеть так:

```
boolean atari(int libertyCount)
    libertyCount < 2
```

Функция эта больше, чем кажется. Математически функция – это множество, одно из подмножеств декартового произведения множества ее входных параметров и множества возможных значений. В данном случае множество входных значений – это `int`, а выходных – `boolean`. И для Java мы получим

```
2L*(Integer.MAX_VALUE+(-1L*Integer.MIN_VALUE)+1L)
```

или 8 589 934 592 элемента в множестве int x boolean. Половина его будет нашим подмножеством, определяющим функцию, поэтому чтобы на 100% убедиться, что функция работает, нам нужно будет протестировать около 4.3×10^9 примеров.

Это показывает суть того, что тесты не могут гарантировать отсутствие ошибок. Тесты могут лишь подтвердить наличие функциональности. Но вернемся к размеру функции. Здесь нам может помочь предметная область. Суть игры Го такова, что количество свободных мест у камня – это не произвольный int, а всегда одно из следующих значений: {1,2,3,4}. Поэтому мы можем переписать функцию:

```
LibertyCount = {1,2,3,4}
boolean atari(LibertyCount libertyCount)
    libertyCount == 1
```

Это сильно меняет дело. Теперь наша функция – множество из всего лишь восьми элементов. Четыре проверки смогут гарантировать, что функция работает верно. Вот почему стоит использовать типы из предметной области вместо встроенных обычных. Использование типов предметной области практически всегда позволяет сделать функции меньше (в вышеописанном смысле). Один из способов найти эти типы – это подобрать пример проверки в терминах предметной области до написания функции.

Автор оригинала - Keith Braithwaite

Пишите тесты для людей

(В оригинале - Write Tests for People)

Вы пишете автоматизированные тесты для своего кода? Мои поздравления! Вы их пишете до написания кода? Еще больше поздравлений! Это действие сразу выводит вас в сторонников самых передовых практик. Но насколько хорошие тесты вы пишете? Как вы можете это оценить? Один из способов – это спросить себя «Для кого я пишу тесты?». Если ответом будет «Для себя, чтобы уменьшить время на поиски ошибок», то скорее всего, ваши тесты – не так хороши, как могли бы быть. Так для кого нужно писать тесты? Для тех, кто будет пытаться разобраться в вашем коде.

Хорошие тесты работают как документация к коду, который они тестируют. Они описывают, как именно код работает. Для каждого тестового сценария они:

1. Описывают контекст, стартовую точку или предусловия, которые должны быть выполнены.
2. Демонстрируют, как программа должна вызываться.
3. Описывают ожидаемые результаты или постусловия, которые должны быть проверены.

Различные сценарии будет слегка отличаться друг от друга. Тот, кто будет разбираться в коде, должен иметь возможность посмотреть на несколько тестов и, сравнив эти три части, увидеть, что именно заставляет программу работать по-другому. Каждый тест должен ясно показывать причины и эффект от взаимодействий этих трех частей. Это показывает, что непосредственно невидимое в тесте не менее важно того, что видно. Слишком много кода в тесте будет отвлекать читателя на неважные детали. Старайтесь скрывать такие детали где только можно при помощи, например, функций с понятными именами – рефакторинг Extract Method вам в этом поможет. Также убедитесь, что каждый тест имеет адекватное имя, описывающее используемый сценарий так, чтобы пользователю не пришлось копаться в коде, чтобы понять, что же этот тест делает. Кроме этого, имя тестового класса и его методов должны включать как минимум стартовую точку и то, как программа запускается. Это позволит оценить покрытие тестами простым просмотром имен методов класса. В принципе неплохо включить в имя и ожидаемый результат, но только если в результате имя не получится слишком длинным.

Хорошей идеей является протестировать ваши тесты. Вы можете проверить, что они действительно обнаруживают ошибки путем помещения такой ошибки в программе (конечно же, делать это надо только в своей локальной копии, которую вы сразу же

выбросите после испытания!). Убедитесь, что сообщения об ошибках выдаются в понятном и значимом виде. А также проверьте, что выдаваемые тестами сообщения понятны тем, кто пытается понять ваш код. Единственный способ это проверить – найти кого-нибудь, кто не знает вашего кода, показать ему тесты, попросить их просмотреть и спросить, что он в результате понял. Внимательно выслушайте ответ. Если он что-то не сможет понять, то скорее всего это не потому, что он недостаточно старался, а потому, что вы это сделали недостаточно понятным.

Автор оригинала - Gerard Meszaros

Планируйте свой следующий коммит

(Автор оригинала - Know Your Next Commit)

Я спросил у трех программистов, что они делают в настоящий момент. Один ответил «Я делаю рефакторинг вот этих методов», второй сказал «Я добавляю новые параметры в этот web action», третий ответил «Я работаю над этим требованием пользователя».

Может показаться, что первые два углубились в детали, и только третий может видеть общую картину, соответственно, именно он делает правильнее. Однако, когда я спросил их, когда и что они будут коммитить, картина резко поменялась. Первые два очень точно сказали, какие файлы они меняют и что закончат где-то через час-два. Третий же ответил «Ну, я думаю закончить где-то через несколько дней. Возможно, я добавлю пару новых классов и может быть поменяю этот сервис».

Первые два не потеряли видения общей картины. Они выбрали задачи, ведущие по их мнению в правильном направлении, так, чтобы закончить их за несколько часов. По окончанию этих задач они бы выбрали следующие, и так далее. В результате весь код был бы написан с ясной целью, достижимой за достаточно короткое время.

Третий же программист не был способен разбить задачу на части и пытался решить ее целиком. У него не было идей, что он будет делать, и он в общем-то занимался тем, что можно назвать «умозрительным программированием», надеясь рано или поздно получить что-то работающее и пригодное для коммита. Очень вероятно, что код, написанный в начале этого процесса, не слишком хорошо будет соответствовать окончательному решению, принятому в результате.

Что бы делали первые два программиста, если бы оказалось, что их задача требует более двух часов? Поняв, что времени потребуется намного больше, они бы отбросили то, что уже сделано, определили бы меньшие задачи, и начали бы их решать. В противном случае они бы тоже могли углубиться в умозрительные процессы, получив код с более низким качеством в репозитории. Вместо этого лучше пожертвовать уже сделанными изменениями, но сохранить полное понимание процесса.

Третьему же программисту не оставалось бы ничего другого, кроме как безрассудно пытаться слепить свои изменения воедино, получив что-то пригодное для коммита. Ему бы пришлось выбрасывать слишком много кода, чтобы вернуться к началу, ведь так? К сожалению, альтернатива этому – плохо продуманный код в репозитории, написанный без ясного понимания цели.

В какой-то момент даже сфокусированные на конкретных задачах программисты могут потерпеть неудачу, углубясь в более длительные умозрительные процессы, однако потом, вновь вернувшись к пониманию цели, отбросить не слишком качественный код, переписав его уже согласно пришедшему озарению. Т.е. даже у такого шага в результате есть цель – найти способ решить задачу продуктивно.

Планируйте свой следующий коммит. Если у вас не получается закончить текущую задачу, отбросьте все и начните сначала, взявшись за другую задачу. При необходимости погружайтесь в свободные умозрительные эксперименты, но делайте это осознанно. Не помещайте свои предположения в репозиторий.

Автор оригинала - Dan Bergh Johnsson

Поддерживайте чистоту кода

(В оригинале - Keep the Build Clean)

Вы когда-нибудь смотрели на длинный-длинный список предупреждений компилятора, выдаваемого в случае плохо написанного кода, думая про себя «С этим точно надо что-то делать… но сейчас на это совсем нет времени». И другая ситуация – вы смотрите на единственное только что появившееся предупреждение… и просто его исправляете.

Когда я начинаю новый проект с нуля, то в нем нет предупреждений вообще. Но по мере написания кода, если не обращать на предупреждения внимания, то их количество начинает расти (а вместе с ним и количество проблем). Когда в сообщениях компилятора много шума, гораздо сложнее найти важное сообщение среди сотен неважных.

Чтобы предупреждения снова стали осмысленными, я стараюсь делать так, чтобы их вообще не было. Даже если предупреждение неважное, я его устраняю. Если оно сообщает о потенциальной проблеме, пусть я даже уверен, что она не произойдет, я ее исправлю. Например, если компилятор предупредит о возможном использовании неинициализированного указателя, то проще это исправить, даже если в реальности эта ситуация не наступит. Если система генерации документации сообщает о незадокументированном параметре – я добавлю этот комментарий.

Если что-то становится реально неактуальным, я советуюсь с командой по поводу изменения политики генерации предупреждений. Если, например, я нахожу документирование параметров бесполезным занятием, то надо сделать так, чтобы предупреждения в этих случаях не выдавалось вообще. Или, например, при переходе на новую версию языка или компилятора старый код, ранее не выдававший предупреждений, начнет их выдавать в большом количестве. Часто исправление таких предупреждений не будет иметь смысла, а их наличие тоже ничего хорошего не принесет.

Поддерживая код чистым, я не трачу времени на анализ того, является ли то или иное предупреждение важным или нет. Такой анализ требует времени и усилий, и исключение его все лишь упростит. Чистый код также проще передать кому-то на сопровождение. Если в коде есть предупреждения, то кому-то придется их все просмотреть, чтобы определить важные и неважные. Или, что более вероятно, просто их все проигнорировать, включая и важные.

Предупреждения компилятора важны. Вам лишь нужно устранить шум, чтобы начать ими пользоваться. Не откладывайте, как только появляется новое предупреждение, сразу же его исправляйте. Или же отключайте его в компиляторе, меняя политику их генерации. Чистый код – это не только отсутствие ошибок. Предупреждения – тоже важная часть гигиены кода.

Автор оригинала - Johannes Brodwall

"Подмоченный" код сложнее оптимизировать

(В оригинале - WET Dilutes Performance Bottlenecks)

Важность принципа DRY (Не повторяйся, Don't Repeat Yourself) в том, что он реализует идею, согласно которой каждый фрагмент знаний представлен лишь единожды. Другими словами, каждое знание должно быть реализовано один и только один раз. Противоположность ему – принцип WET (Реализуй каждый раз, Write Every Time). (В английском здесь игра слов – аббревиатуры WET и DRY имеют еще и смысловое значение «мокрый» и «сухой»). WET код – код, в котором знание закодировано в нескольких реализациях. Влияние DRY и WET на производительность станет ясным после рассмотрения их влияния на профилирование.

Давайте предположим, что одна из функций нашей системы, скажем, *X*, является «узким местом» по производительности. Пусть, например, она расходует 30% процессорного времени. И теперь предположим, что она реализована в коде в десяти различных местах. В среднем, каждая реализация будет тратить около 3% процессорного времени. И вот тут мы можем просто не заметить, что данная функциональность является узким местом. Однако если мы даже каким-то образом и распознали узкое место, нам потребуется найти и исправить все десять реализаций. Будь наш код написан по принципу DRY, мы бы, во-первых, ясно увидели, что на функциональность расходуется 30% процессорного времени, а во-вторых, исправлять бы нам пришлось тоже лишь единственное место в коде. И да, нам бы не надо было тратить время на то, чтобы сначала найти все десять мест, как это было бы нужно в WET коде.

Есть один общий случай, когда часто нарушается принцип DRY – это использование множеств. Общей техникой реализации запроса является проход по всему множеству и применение запроса к каждому из элементов:

```
public class UsageExample {
    private ArrayList<Customer> allCustomers = new ArrayList<Customer>();
    // ...
    public ArrayList<Customer> findCustomersThatSpendAtLeast(Money amount) {
        ArrayList<Customer> customersOfInterest = new ArrayList<Customer>();
        for (Customer customer: allCustomers) {
            if (customer.spendsAtLeast(amount))
               customersOfInterest.add(customer);
        }
        return customersOfInterest;
    }
}
```

Открывая это множество клиентам, мы нарушаем инкапсуляцию. И это не только ограничивает нас в возможностях рефакторинга, а также вынуждает пользователей нашего кода нарушить DRY, поскольку наиболее вероятно, что каждому из них придется еще раз реализовать практически аналогичный запрос. Ситуации легко избежать, убрав «сырое» множество из API. В данном случае мы предоставим новый, специфичный для предметной области тип `CustomerList`. Данный класс послужит «домом» для всех наших запросов.

Этот новый тип легко позволит нам отслеживать факт, являются ли запросы «узким местом» или нет. Включив запросы в класс, мы избавимся от необходимости открывать представления вроде `ArrayList` клиентам. Это дает нам свободу изменять реализации без страха нарушить контракт с клиентом:

```
public class CustomerList {
    private ArrayList<Customer> customers = new ArrayList<Customer>();
    private SortedList<Customer> customersSortedBySpendingLevel = new SortedList<Customer
    // ...
    public CustomerList findCustomersThatSpendAtLeast(Money amount) {
        return new CustomerList(customersSortedBySpendingLevel.elementsLargerThan(amount)
    }
}

public class UsageExample {
    public static void main(String[] args) {
        CustomerList customers = new CustomerList();
        // ...
        CustomerList customersOfInterest = customers.findCustomersThatSpendAtLeast(someMi
        // ...
    }
}
```

В данном примере следование принципу DRY позволило нам предоставить альтернативную схему индексирования при помощи отсортированного списка. Но более важно то, что следование принципу DRY помогло нам найти и устранить узкое место в производительности, что было бы труднее, будь код написан по принципу WET.

Автор оригинала - Kirk Pepperdine

Подозреваете ошибку в компиляторе? Проверьте получше свой код!

(В оригинале - Check Your Code First before Looking to Blame Others)

Все разработчики часто склонны не верить, что ошибка в их коде, обвиняя вместо этого, например, компилятор.

Однако в реальности то, что ошибка на самом деле в компиляторе, интерпретаторе, операционной системе, сервере приложений, базе данных, менеджере памяти или еще где-нибудь среди системного ПО, весьма маловероятно. Да, такие ошибки тоже есть, однако случаются они гораздо реже, чем бы нам хотелось.

Один раз я сталкнулся с реальной ошибкой компилятора, неправильно оптимизировавшего цикл, однако я обвинял компилятор и операционку гораздо чаще. Я убил очень много времени на попытки доказать наличие ошибок в системе, и все лишь для того, чтобы в конце испытать неловкость, найдя ошибку в своем коде.

Если инструмент широко распространен и используется множеством людей в различных ситуациях, то причин сомневаться в его качестве очень мало. Конечно, если инструмент еще в стадии раннего релиза, используется лишь несколькими людьми в мире, имеет версию 0.1 или же относится к редко скачиваемому open source, то основания для подозрений могут быть гораздо более весомыми. То же самое справедливо и для альфа-версий известных коммерческих продуктов.

Взяв за основу то, что ошибки в компиляторах – очень большая редкость, вы сможете гораздо эффективнее тратить свое время на поиски ошибок у себя вместо попыток доказать, что виноват компилятор. Применяйте все обычные практики отладки: изолируйте проблему, отслеживайте вызовы, окружайте ее тестами, проверяйте соглашение о вызовах, общие библиотеки и версии, расскажите о проблеме кому-нибудь еще, проверьте переполнение и повреждение стека, запустите код на разных машинах и с разными конфигурациями сборки (debug и release).

Если кто-то заявляет о проблеме, которую вы не можете воспроизвести, то пойдите и посмотрите, как это у него получается. Возможно, он делает что-то, что вы не учли, или делает это каким-то другим способом.

Моим правилом стало то, что если я сталкиваюсь с ошибкой, которую не могу найти, и у меня появляются мысли, что ошибка в компиляторе, то самое время поискать повреждения стека. Особенно если добавление отладочного кода приводит к исчезновении ошибки.

Проблемы в многозадачных системах – еще один тип ошибок, способный увеличить количество седых волос у вас и бросить тень подозрения на компилятор или ОС. Все рекомендации поддерживать код простым становятся в разы актуальнее, когда вы работаете с многозадачной системой. Отладка и модульное тестирование практически не способны отлавливать такие ошибки, поэтому простота кода выходит на первое место.

Так что перед тем, как обвинить компилятор, вспомните правило Шерлока Холмса: «После того, как вы исключили невозможное, то все, что осталось, независимо от того, насколько невероятным оно выглядит, является истиной», и отдавайте предпочтение именно ему, а не правилу Дирка Джентли «Посл того, как вы исключили невероятное, все, что осталось, независимо от того, насколько невозможным оно выглядит, является истиной».

Автор оригинала - Allan Kelly

Позвольте трупу упасть

(В оригинале - Don't Nail Your Program into the Upright Position)

> Однажды я написал шутливую программку на C++, в которой я с некоторой долей сатиры реализовал следующую концепцию: при помощи множества try...catch конструкций по всему коду не дать программе аварийно завершиться. Результат мы назвали «труп, не падающий потому, что он прибит гвоздями к стене».

Несмотря на легкомысленность примера, я вынес из него некоторый опыт.

В нашей собственной C++ библиотеке был один класс. Этот класс долгое время страдал от активности многих и многих программистов. Практически каждый оставил в нем свой след. В этом классе содержался код, работающий со всеми исключениями из всех остальных мест. И мы решили, что объект этого класса должен жить вечно (или умереть в попытках это сделать).

Для этого мы сплели сеть из множества обработчиков исключений. Мы смешали в кучу как «родные» обработчики, так и обработчики Windows. Когда что-то не работало, мы вызывали это снова и снова. Оглядываясь назад, мне нравится сравнивать написание конструкций try..catch, вложенных в другие ветви catch, с решением свернуть с широкой дороги проверенных практик на манящую, но опасную для здоровья узкую тропинку, ведущую непонятно куда. Однако, это уже «здравый смысл постфактум».

Не стоит и говорить, что когда в приложении, использующем этот класс, что-то шло не так, то это что-то исчезало ничуть не хуже, чем жертвы мафиозной разборки в порту, не оставляя даже пузырей на поверхности. И даже не вызывая предназначенных для этого отладочных функций, цель которых – записывать информацию о случающихся проблемах. В итоге мы конечно поняли, какую херню мы сделали, и пришли в ужас. Мы заменили этот кошмар на простой и надежный механизм оповещения. Но произошло это после многих и многих сбоев, оставшихся необнаруженными.

Я бы не стал об этом писать, если бы не недавно состоявшийся диалог с челом, чей академический статус должен был говорить «Я знаю лучше». Мы обсуждали Java-код в удаленной транзакции. Он настаивал, что в случае ошибки код должен перехватить исключение на своей стороне. «И что ему потом с ним делать?» - спросил я, – «Приготовить на ужин?»

Он процитировал правило дизайнера пользовательского интерфейса: «НИКОГДА НЕ ПОЗВОЛЯЙТЕ ПОЛЬЗОВАТЕЛЮ ВИДЕТЬ СООБЩЕНИЕ ОБ ОШИБКАХ», не задумываясь о том, что КАПС и НИКОГДА – не самые лучшие аргументы. Я спросил,

не он ли писал софт для банкоматов, «синий экран» которых так любят фоткать и выкладывать в инет, за что получил по голове.

В общем, если вы с ним встретитесь, то улыбайтесь и соглашайтесь со всем, что он говорит, пока будете медленно продвигаться к двери.

Автор оригинала - Verity Stob

Помещайте все в систему контроля версий

(в оригинале Put Everything Under Version Control)

Помещайте все, что относится к вашему проекту, в систему контроля версий. Все, что вам для этого нужно, это: бесплатные инструменты, такие как SVN, Git, Mercurial и CVS; достаточное количества места на диске; недорогой производительный сервер; доступ к сети; или же, как вариант, можно воспользоваться сервисом хостинга проектов.

После того, как вы установите систему контроля версий себе на компьютер, все, что нужно сделать, чтобы поместить ваш проект в репозиторий – это выполнить определенную команду в папке, содержащей «чистый» проект. Вам нужно будет выучить лишь две новых операции – коммит (помещение изменения в репозиторий) и апдейт (забирание последних изменений из репозитория в свою рабочую копию).

Как только ваш проект помещен в систему контроля версий, вы можете легко отслеживать его историю, видеть, кто что написал, а также ссылаться на любую версию любого файла проекта. Но что гораздо важнее, вы теперь смело можете вносить изменения! Никакого кода, закомментированного на случай того, что он понадобится в будущем – он отлично сохранится в предыдущей версии в репозитории. Вы можете (и должны) помечать релизы тегами с читаемыми именами, чтобы в будущем иметь возможность быстро получить точно такую же версию, которая стоит у вашего клиента. Вы можете создавать ветви параллельной разрабоки – большинство проектов имеют главную ветку и одну или более веток для поддержки старых версий, имеющихся у клиентов.

Система контроля версий минимизирует количество «трения» между разработчиками. Когда программисты работают над разными частями проекта, интеграция происходит просто волшебным образом. Если же их работа пересекается, система это обнаруживает и позволяет вручную решить возникшие конфликты. При дополнительных настройках система может информировать всех участников о каждом сделанном коммите, способствуя общему видению прогресса.

Помещая проект в систему контроля версий, не старайтесь сэкономить – помещайте туда все, что можно. Кроме исходных кодов, добавьте документацию, инструментарий, скрипты сборки, тестовые сценарии и библиотеки. Помещение всего проекта в (регулярно резервнокопирующийся) репозиторий значительно снижает риск потери данных из-за отказа вашего жесткого диска. А установка среды разработки на новом

рабочем месте превращается в простое извлечение всего проекта из репозитория. Это облегчает сборку и тестирование кода на разных платформах – на любом компьютере команда «апдейт» обеспечит вам последнюю версию проекта.

После первого знакомства с красотой работы с системой контроля версий следование нескольким простым правилам еще более повысит эффективность вашей команды:

- Помещайте отдельно каждое логически законченное изменение. Сваливание нескольких изменений в один коммит усложнит вычленение отдельного изменения в дальнейшем. Особенно в случае проведения серьезного рефакторинга или изменения стиля – они легко могут скрыть другие изменения.
- Сопровождайте каждый коммит поясняющим сообщением. Как минимум, кратко опишите, что вы сделали, но если у вас есть какие-либо пояснения к сделанному, то комментарий к коммиту – лучшее для них место.
- И наконец, избегайте помещать в репозиторий код, «ломающий» сборку, в противном случае вы быстро станете непопулярными среди ваших коллег.

Жизнь с использованием систем контроля версий слишком хороша, чтобы ее разрушать неправильными действиями.

Автор оригинала - Diomidis Spinellis

Послание в будущее

(В оригинале - A Message to the Future)

В течении многих лет я обучала программистов, и за эти годы у меня сложилось впечатление, что большинство из них мыслят примерно следующим образом: «Раз я решаю очень сложную проблему, то и ее решение должно быть столь же сложным для понимания для всех остальных, в том числе и для меня, если я вдруг решу взглянуть на него спустя пару месяцев»

Я помню один случай со своим студентом Джо, пришедшим однажды показать мне то, что он написал, со словами «Держу пари, вы не догадаетесь, что делает этот код!».

«Да, ты прав», - сказала я, не вдаваясь в подробности написанного, а вместо этого задумавшись, как бы лучше донести до него то, что я хочу сказать. «Я уверена, ты старался, делая этот код. Но я не уверена, что ты при этом не забыл об одной важной вещи. Джо, у тебя ведь есть младший брат?»

«Да, конечно есть! Его зовут Фил, и он как раз пошел на ваш вводный курс. Он тоже изучает программирование!» - гордо ответил Джо.

«Отлично!» - ответила я. «Я вот только не уверена, что он сможет разобраться в твоем коде».

«Ни шанса!» – ответил Джо. – «Это слишком сложно!».

«А попробуй представить, - сказала я, - что этот код – реальный, работающий, и спустя несколько лет твоего брата наймут слегка этот код дописать. Что ты сделал для него в этом случае?» Джо лишь смотрел на меня, моргая. «Мы оба знаем, что Фил весьма способный, не так ли?» Джо кивнул. «И хотя я и не люблю это говорить, я тоже достаточно способная». Джо улыбнулся. «Итак, если я не могу разобраться, что делает этот код, и твой брат скорее всего тоже будет озадачен этим, то что же в итоге ты написал?». Джо посмотрел на свой код слегка с другой стороны, как мне показалось. Я постаралась придать голосу максимальную доброжелательность: «Попробуй посмотреть на каждую строчку кода как на сообщение кому-то в будущем. Кому-то, кто может оказаться твоим младшим братом. Представь, что ты объясняешь этому способному преемнику то, как же решена эта сложная задача».

«Можешь это представить? Чтобы этот способный программист из будущего увидел твой код и сказал: «Вау! Великолепно! Я понимаю все, что здесь написано и я впечатлен тем, насколько это сделано элегантно. Я должен показать это всем в команде! Это писал настоящий мастер!»».

«Джо, как ты думаешь, мог бы ты написать код, который бы и задачу решал, и при этом был бы столь элегантным? Который бы звучал как красивая мелодия? Я думаю, что вы все, решающие разные задачи, могли бы написать столь же элегантный код, не так ли? Может, мне стоит начать оценивать код за красоту? Что ты думаешь, Джо?»

Джо посмотрел на меня, по его лицу гуляла улыбка. «Я понял, профессор, я постараюсь сделать этот мир лучше для Фила. Спасибо!».

Автор оригинала - Linda Rising

Правило туриста

(В оригинале - The Boy Scout Rule)

У настоящих туристов есть правило: «Всегда оставляй место стоянки чище, чем оно было до вас». Если стоянка загажена, вы ее очищаете независимо от того, кто ее загадил до вас. Тем самым вы обеспечиваете приятное окружение следующей группе туристов. В оригинале это правило, сформулированное основоположником движения скаутов Робертом Стивенсоном Смитом Баден-Пауэлом, звучало так: «Старайтесь оставить этот мир чуть лучше, чем он был до вашего прихода»

Что, если мы будем следовать этому правилу в нашем коде? «Всегда помещай назад код, измененный хоть чуть к лучшему, чем ты его брал». Неважно кто автор кода, просто взять и потратить чуть-чуть времени на улучшение. Что получится в результате?

Я думаю, если бы мы все следовали этому простому правилу, то очень скоро мы бы стали свидетелем конца непрекращающемуся ухудшению программных систем. Вместо этого бы наши системы постепенно становились лучше и лучше. А команды бы работали над улучшением систем в целом, а не каждый над своей индивидуальной частью.

Я не считаю, что это правило слишком сложно для выполнения. Не нужно доводить код до идеала перед помещением. Всего лишь чуть-чуть лучше, чем он был. Разумеется, это подразумевает, что добавляемый вами код также должен быть отличным. И это подразумевает, что при каждом помещении вы улучшаете что-то еще. Переименовать неочевидную переменную, разбить длинную функцию на две маленьких. Разорвать циклическую зависимость. Или еще что-нибудь в том же духе.

Откровенно говоря, это должно стать правилом приличия, как мытье рук после туалета или выбрасывание мусора в урну, а не куда попало. Написание «мусорного» кода должно быть столь же неприемлемым, чем-то таким, чего никто не делает.

Но есть еще одно «но». Кроме заботы о своем собственном коде, нужно еще заботиться и коде всей команды. Команда должна помогать друг другу, в том числе и убирать мусор друг за другом. Следование правилу настоящего туриста выгодно всем!

Автор оригинала - Uncle Bob

Предотвращайте ошибки

(В оригинале - Prevent Errors)

Сообщения об ошибках – наиболее критическое взаимодействие между пользователем и системой. Они появляются в тот момент, когда взаимодействие между пользователем и системой уже «на пределе».

Легче всего думать, что ошибки – это результат неправильного ввода данных от пользователя. Однако люди делают одни и те же ошибки. Поэтому возможно «отладить» коммуникацию пользователя с системой точно также, как вы отлаживаете остальные части системы.

Например, вы хотите, чтобы пользователь ввел дату из разрешенного диапазона. Вместо того, чтобы позволять пользователю вводить дату непосредственно, лучше использовать календарь или выпадающий список, в котором будут только даты из разрешенного диапазона. Таким образом ввести неправильную дату будет просто невозможно.

Ошибки форматирования – еще одна общая проблема. Например, если пользователю предоставлено текстовое поле для ввода даты, и он вводит туда «29 июля 2012 года», то не очень хорошо не принимать этот ввод только потому, что он не в формате «дд \мм\гггг». А еще хуже отвергать «29 \ 03 \ 2012» из-за лишних пробелов – пользователю будет сложно понять, в чем причина, ведь дата вводится в запрашиваемом формате.

Ошибки такого типа возникают из-за того, что гораздо проще выдать сообщение об ошибке, чем рассмотреть три или четыре наиболее распространенных варианта ввода. Подобные «ошибки» приводят к раздражению пользователей, что приводит к новым ошибкам из-за потери концентрации. Вместо этого уважайте желание пользователя вводить информацию, а не данные.

Еще один способ избежать ошибок форматирования – подсказать пользователю, чего от него ждут. Например, указав ожидаемый формат «дд\мм\гггг». Еще одной подсказкой может быть разделение поля ввода на три отдельных поля длиной в два, два и четыре символа.

Подсказки отличаются от инструкций. Подсказки появляются в момент взаимодействия, инструкции – перед ним. Подсказки предоставляют контекст, инструкции диктуют поведение.

В большинстве случаев инструкции неэффективны в предотвращении ошибок. Пользователь ожидает, что интерфейс будет работать привычным для него образом (Ведь каждый знает, что значит «29 июля 2012 года», не так ли?). Поэтому инструкции никто не читает. А подсказки подталкивают пользователя к правильному варианту.

Еще один способ предотвращения ошибок – это предложить вариант по умолчанию. Например, пользователь чаще всего вводит даты «сегодня», «завтра», «день рождения», «дедлайн» или дату, вводимую в эту форму в прошлый раз. В зависимости от контекста можно с высокой вероятностью выбрать правильный вариант по умолчанию.

Системы должны позволять пользователю ошибаться. Вы можете это сделать, реализовав множественную отмену всех действий, особенно тех, которые могут потенциально уничтожить данные пользователя.

Логирование и анализ отмен может выявить места наиболее частых ошибок, таких как постоянное нажатие на одну и ту же «неправильную» кнопку. Такие ошибки могут возникать из-за неправильных подсказок, которые лучше переделать для предотвращения дальнейших ошибок.

Большинство ошибок – систематические, результат непонимания между пользователем и программным обеспечением. Понимание того, как пользователь думает, интерпретирует информацию, принимает решения и вводит данные, поможет вам отладить коммуникацию между вашим программным обеспечением и пользователем.

Автор оригинала - Giles Colborne

Применяйте принципы функционального программирования

(В оригинале - Apply Functional Programming Principles)

Не так давно в сообществе программистов стал возрождаться интерес к функциональному программированию. Одна из причин – то, что функциональная парадигма легко может использовать развитие мультиядерности, к которой движется современная ИТ индустрия. Однако, хотя это и важный фактор, причина для написания этой статьи, советующей пристальнее взглянуть на функциональное программирование – другая.

Владение функциональным программированием может сильно улучшить качество вашего кода, написанного в остальных контектстах. Если вы хорошо понимаете и применяете функциональную парадигму, то в вашем дизайне будет преобладать чистые функции (referential transparency).

Чистые функции – очень желательная вещь. Это подразумевает, что функции всегда возвращают одни и те же результаты для одних и тех же входных данных, независимо от того, где и когда они были вызваны. То есть, выполнение функций слабо зависит (в идеале – вообще не зависит) от изменяемого состояния.

Одна из ведущих причин появления дефектов в процедурном программировании – изменяемые переменные. Наверняка каждый читающий эту статью не раз искал причину, почему значение какой-либо переменной не такое, как должно быть в определенной ситуации. Области видимости могут смягчить эти скрытые дефекты или по крайней мере серьезно сузить область их появления, однако при этом они же способствуют появлению дизайна с чрезмерной изменяемостью.

И мы к сожалению ничего не можем ожидать от текущего положения вещей в индустрии. ООП неявно продвигает подобный подход к дизайну, потому что множество примеров представляют из себя графы из объектов, вызывающих изменяющие методы друг друга. Однако при помощи дизайна через тестирование (test-driven design) можно избавиться от ненужной изменяемости.

В результате получится дизайн с гораздо лучшим разделением ответственности, с множеством небольших функций, зависящих только от переданных аргументов вместо использования изменяемых переменных. Это приведет к меньшему количеству дефектов и более простой отладке, потому что при таком дизайне определить, когда значение стало неправильным, намного проще. И все это – лишь из-за более высокой

степени чистоты функций, и скорее всего ничего не вобьет эти идеи вам в голову лучше, чем изучение функционального языка программирования, в котором такая модель вычислений – норма.

Разумеется, функциональный подход не оптимален для всех возможных случаев. Например, в объектно-ориентированных системах этот стиль обычно приносит гораздо лучшие результаты для разработки бизнес-модели, а не для разработки пользовательского интерфейса.

Изучите функциональное программирование настолько, чтобы быть способным применить полученные знания в других областях. Объектно-ориентированный подход может быть гораздо ближе к своей противоположности – функциональному подходу, чем это принято считать. Возможно, вы даже найдете их практически зеркальным отражением друг друга, подобно компьютерным инь и ян :)

Автор оригинала - Edward Garson

Принцип единственности ответственности

(В оригинале - The Single Responsibility Principle)

Один из основных принципов хорошего дизайна – это:

> Помещай вместе то, что изменяется по одной и той же причине, и разделяй то, что изменяется по разным причинам.

Этот принцип часто называют «принципом единственности ответственности». Говоря коротко, принцип декларирует, что подсистема, модуль, класс или даже функция не должны иметь более одной причины для изменения своего кода. Классический пример – класс для работы с бизнес-правилами, отчетами и базой данных:

```
public class Employee {
  public Money calculatePay() ...
  public String reportHours() ...
  public void save() ...
}
```

Некоторые программисты могут считать, что помещение этих трех функций в общий класс – отличная практика. В конце концов, класс и является коллекцией функций, обрабатывающих общие данные. Однако, проблема здесь в том, что изменения в эти три функции нужно будет вносить по полностью различным причинам. Функция calculatePay будет меняться, если изменится бизнес-правило начисления зарплаты. Функция reportHours будет меняться, если кому-то понадобится другой формат отчета. А функция save изменится, если изменится схема БД. И эти три причины вместе делают класс Employee сильно изменчивым. Он будет меняться по любой из них. Более того, эти изменения коснутся и всех классов, зависящих от класса Employee.

Хороший дизайн системы означает, что мы разделяем систему на компоненты, каждый из которых может быть выпущен независимо. Независимый выпуск означает, что если мы изменим только один компонент, то нам не придется перевыпускать все остальные. Однако, если в данном примере класс Employee активно используется другими классами, то его изменение повлечет за собой необходимость перевыпуска и остальных компонентов тоже. Это разрушает все преимущества компонентного дизайна (еще один используемый термин – сервис-ориентированная архитектура, SOA).

```
public class Employee {
  public Money calculatePay() ...
}

public class EmployeeReporter {
  public String reportHours(Employee e) ...
}

public class EmployeeRepository {
  public void save(Employee e) ...
}
```

Приведенное выше разбиение решает эту проблему. Каждый из классов может быть помещен в отдельный компонент. Точнее, все классы, работающие с отчетом, помещаются в компонент отчета, все классы, работающие с БД – в компонент БД, а все, что касается бизнес-правил – в компонент бизнес-правил.

Внимательный читатель может заметить, что в приведенном выше примере все еще есть зависимости. Так, класс Employee все еще зависит от остальных классов. Т.е. если изменится именно класс Employee, то скорее всего, остальные классы нужно будет перекомпилировать и перевыпустить тоже. Таким образом, класс Employee по-прежнему не может быть изменен и выпущен независимо. Однако, остальные классы могут меняться и перевыпускаться независимо. Изменение одного из них не повлечет перекомпиляции и перевыпуска остальных. И даже Employee можно сделать независимо выпускаемым, аккуратно используя принцип инверсии зависимости, однако это уже совсем другая тема.

Аккуратное применение принципа единственности ответственности, разделяя вещи, меняющиеся по разным причинам – ключевой фактор для создания дизайна, в основе которого лежит структура отдельно выпускаемых компонент.

Автор оригинала - Uncle Bob

Программирование - это дизайн

(В оригинале - Code Is Design)

Представьте, что вы проснетесь завтра и обнаружит, что строительная индустрия непостижимым образом совершила скачок вперед. Миллионы дешевых и быстрых роботов могут изготавливать материалы из воздуха, не требуют внешнего источника энергии для работы и могут сами себя ремонтировать. Если им выдать непротиворечивый план строительства, эти роботы могут выполнить работу самостоятельно, без участия человека, с весьма незначительной себестоимостью.

Представить результат такого воздействия на индустрию строительства можно, но что будет дальше? Как изменится поведение архитекторов и дизайнеров, если стоимость самой конструкции будет незначительной? Сегодня перед началом реального строительства строятся и тщательно тестируются физические и компьютерные модели. Будет ли нужно тратить столько сил, если себестоимость строительства будет практически нулевой? Если постройка разрушится – ничего страшного, нужно лишь найти, что было не так и пусть роботы строят еще одну, с учетом найденной проблемы. Есть и еще одно следствие. Без использования моделей незаконченный дизайн будет развиваться путем повторяющегося строительства до получения необходимого результата. И случайный наблюдатель с трудом сможет отличить незаконченный дизайн от законченного.

Возможность прогнозировать сроки практически исчезнет. Стоимость строительства гораздо проще расчитать, чем стоимость дизайна. Мы примерно знаем, сколько стоит один кирпич, и сколько кирпичей нам понадобится. Если же предсказуемая часть задачи будет стремиться к нулю, на первый план выйдет непредсказуемая – дизайн. Результаты будут получаться быстрее, но за практически непредсказуемое время.

И конечно же, давление конкурентной экономики никуда не денется. При стоимости строительства, близкой к нулю, компания, закончившая дизайн раньше, раньше и выйдет на рынок. Сделать дизайн как можно быстрее – это станет главным фактором давления на фирмы, занимающиеся проектированием. И за этим неизбежно произойдет и то, что кто-то, глядя на то, что неспециалист не сможет отличить законченный проект от незаконченного, и глядя на преимущества быстрого выхода на рынок, скажет: «Хватит, и так сойдет».

Отдельные жизненно важные проекты будут делаться более основательно, но в большинстве случаев заказчикам придется научиться жить с незаконченным дизайном. Компании ведь всегда смогут послать своих роботов «подлатать»

покосившееся строение. Все говорит о том, что (как бы это не выглядело нелогичным) серьезное снижение стоимости строительства повлечет за собой столь же серьезное снижение качества.

Думаю, не будет сюрпризом, если вы найдете аналогию с программированием. Если мы допустим, что программирование – это дизайн (креативный процесс, а не рутинная работа), то это хорошо объяснит кризис в отрасли. У нас кризис дизайна: требования к качеству дизайна превосходят наши возможности. Принуждение использовать незавершенный дизайн слишком высоко.

К счастью, эта модель также дает подсказку, что делать. Физическая модель – это автоматизированное тестирование. Программа не завершена до тех пор пока не будет протестирована. Усовершенствованные языки и практики дизайна также дают надежду. И наконец, один неизбежный факт: отличный дизайн делается отличными дизайнерами, посвятившими себя этой профессии. И программирование не является исключением.

Автор оригинала - Ryan Brush

Программируйте на языке предметной области

(В оригинале - Code in the Language of the Domain)

Представьте себе два исходных кода. В первом вы видите что-то подобное:

```
if (portfolioIdsByTraderId.get(trader.getId()).containsKey(portfolio.getId())) {...}
```

Вы чешете затылок, пытаясь догадаться, что может делать этот код. Выглядит как взятие ID из объекта trader и использование его для получения другого ID, после чего – поиск еще одного ID из объекта portfolio… Почесывание затылка почему-то не помогает. Вы ищете определение portfolioIdsByTraderId и находите следующее:

```
Map<int, Map<int, int>> portfolioIdsByTraderId;
```

Постепенно вы выясняете, что этот код делает что-то похожее на выяснение того, имеет ли трейдер доступ к портфолио. И конечно же, этот (или похожий на него) фрагмент кода будет повторяться везде, где будет требоваться выяснить, есть ли у трейдера доступ к портфолио.

А в другом исходном коде вы находите вот такую строчку:

```
if (trader.canView(portfolio)) {...}
```

И никакого почесывания затылка не требуется. Вам не нужно знать подробности, что у этой функции внутри. Вероятнее всего, там будет подобное приведенному выше фрагменту, но это уже дело трейдера, а не ваше.

А теперь скажите, с каким кодом вы бы предпочли работать?

Когда-то давно были только простые структуры данных: биты, байты и символы (на самом деле те же байты, но нам больше нравилось считать их буквами, цифрами и знаками). С числами было уже чуть сложнее – наша десятичная система не очень хорошо «ложилась» на двоичную, и поэтому было несколько различных типов с плавающей запятой. Потом появились массивы и строки (на самом деле такие же массивы). Потом – стеки, очереди, хеши, связанные списки и другие замечательные структуры, не существующие в реальном мире. И «программируя», мы тратили

значительную часть ресурсов на то, чтобы поставить в соответствие объекты из реального мира и эти достаточно ограниченные структуры данных. Настоящие же гуру могли даже запомнить то, как именно они это сделали.

Потом появились типы, определенные пользователем! Это серьезно изменило правила игры. Если в вашей области есть сущности «трейдер» и «портфолио», то вы можете смоделировать их при помощи типов данных с именами Trader и Portfolio. Но еще более важно то, что вы можете моделировать отношения между ними, тоже используя термины из предметной области.

Если вы программируете, не используя термины предметной области, то тем самым вы создаете тайное знание о том, что например вот этот int означает идентификатор трейдера, а вот этот – идентификатор портфолио. (И лучше их не перепутать!). А если вы представляете какую-либо концепцию («Некоторые трейдеры не могут просматривать отдельные портфолио») при помощи алгоритма, подобного поиску соответствия в таблице идентификаторов, то вы не даете тем, кому придется с этим разбираться, вообще никакой подсказки.

Программист, пришедший за вами, не будет иметь того тайного знания, которое имели вы. Так зачем делать его тайным? Использовать ключ для поиска другого ключа, используемого для проверки наличия соответствия – не самый распространенный прием. Каким образом кто-то должен догадаться о том, какое именно правило из предметной области тут скрыто?

Используя имена концепций из предметной области, вы даете возможность другим программистам разобраться в коде гораздо более простым способом, чем пытаться найти обратное отображение от ничего не говорящих алгоритмов к концепциям предметной области. Это также означает, что по мере изменения предметной области (а меняться она будет обязательно, это лишь вопрос времени) вам будет гораздо проще вносить соответствующие изменения в код. При хорошо построенной инкапсуляции шансы, что каждое правило реализовано лишь в одном месте, достаточно высоки, и тогда не придется вносить изменение в множество мест.

Программист, пришедший в ваш проект несколькими месяцами позже, будет очень рад такому подходу. И этим пришедшим через несколько месяцев программистом можете быть вы!

Автор оригинала - Dan North

Программируйте осознанно

(В оригинале - Coding with Reason)

Попытка определить корректность кода вручную приводит к формальной проверке, что получается и дольше, чем само написание кода, и с большей вероятностью ошибки в самой проверке, чем в коде. Автоматические инструменты с этой точки зрения более предпочтительны, но их использование не всегда возможно. Здесь будет описано нечто среднее – полуформальное определение правильности.

Основная идея – вдумчиво разделить код на короткие секции – от одной строки (например, вызов функции) до блоков из десятка строк, и аргументировать их правильность. Аргументы должны быть достаточны для убеждения вашего коллеги, играющего роль «адвоката дьявола».

Секции должны подбираться так, чтобы в конце каждой из них «состояние программы» (значение всех «живых» на данный момент объектов) удовлетворяло легко формулируемому критерию, а функциональность этой секции бы легко описывалась одной задачей. Такие состояния конечных точек обобщают концепции вроде пред- и пост-условий для функций и инвариант (тела) для циклов и классов. Старайтесь, чтобы секции были как можно более независимыми друг от друга.

Большинство широко известных «хороших» практик программирования (хотя им не так часто следуют) облегчают построение аргументации. Уже даже от того, что вы задумаетесь об аргументации правильности вашего кода, вы начнете думать в направлении лучшего стиля и структуры кода. И конечно же, соблюдение большинства этих практик можно проверить при помощи статических анализаторов кода:

1. Избегайте операторов goto, поскольку они делают удаленные друг от друга секции сильно связанными.
2. Избегайте изменяемых глобальных переменных, поскольку они делают секции, их использующие, зависимыми.
3. Каждая переменная должна иметь минимально возможную область видимости. Например, локальный объект может создаваться непосредственно перед его использованием.
4. Делайте объекты неизменяемыми там, где только это возможно.
5. Делайте код легко читаемым используя пробелы, как по горизонтали, так и по вертикали. Например, выравнивайте зависимые структуры и разделяйте секции пустыми строками.
6. Делайте код самодокументируемым, выбирая «говорящие» имена, при этом не

слишком длинные.
7. Если вам нужна вложенная секция, сделайте ее функцией.
8. Делайте функции короткими и сфокусированными на единственной задаче. Старое ограничение на 24 строки все еще актуально. Хотя разрешение экрана сильно выросло, способность человека воспринимать информацию осталась той же.
9. Функции должны иметь ограниченное число параметров (скажем, максимум четыре). Заметьте, это не накладывает ограничение на объем данных, которые можно передать. Просто сгруппируйте данные в объект и передавайте его, заодно обеспечив когерентность и связность этих данных.
10. Каждый модуль должен иметь минимально возможный интерфейс. Меньше коммуникаций – меньше потребуется аргументов для доказательства правильности. К примеру, “getter”, возвращающий внутренние данные объекта – потенциальный источник проблем. Вместо запрашивания у объекта данных, чтобы потом что-то с ней сделать, пусть эту работу со своей информацией делает сам объект. Другими словами, инкапсуляция – лучшее средство для «сужения» интерфейса.
11. Для защиты инвариантности объектов “setter”-методы не должны поощряться, поскольку они легко позволяют нарушить инвариантность.

Кроме доказательства правильности кода, его обсуждение даст вам лучшее его понимание. Развитие проницательности выгодно для всех.

Автор оригинала - Yechiel Kimchi

Программист - профессионал

(В оригинале - The Professional Programmer)

Кто же это – программист-профессионал?

Единственная характерная особенность программиста-профессионала – это личная ответственность. Профессионал сам ответственен за свою карьеру, планирование, соблюдение сроков, ошибки и квалификацию. Профессионал не перекладывает ответственность на других.

- Если вы профессионал, вы ответственны за свою карьеру. Вы ответственны за обучение себя. Вы ответственны за то, чтобы идти в ногу со временем и технологиями. Слишком много программистов думают, что обучение – это забота работодателя. Увы, это совсем не так. Вы думаете, врачи так делают? Или адвокаты? Нет, они совершенствуются в свое личное время и за свои личные средства. Они тратят значительную часть своего личного времени на чтение тематических журналов. Они идут в ногу со временем. И вы должны делать также. Отношения между вами и вашем работодателем четко прописаны в контракте. Кратко – они вам платят, вы делаете свою работу хорошо.
- Профессионалы отвечают за свой код. Они не выпускают код до тех пор, пока он не работает. Только подумайте об этом. Как вы можете считать себя профессионалом, если вы собираетесь выпустить код, в котором не уверены? Профессионалы ожидают, что тестеры не найдут ошибок вообще, потому что они сами очень хорошо протестировали код перед помещением. Конечно же, тестеры что-то найдут, ведь никто не совершенен. Но профессиональное стремление – делать так, чтобы им было нечего находить.
- Профессионалы работают в команде. Они берут ответственность за результат всей команды, а не только за свою часть. Они помогают друг другу, обучают друг друга и обучаются друг у друга, и прикрывают друг друга в случае чего. Если у кого-то в команде проблемы, остальные приходят на помощь, зная, что в следующий раз на его месте могут оказаться они сами.
- Профессионалы не приемлют длинных списков ошибок. Огромный список ошибок – это трагедия беспомощности. Действительно, в большинстве проектов необходимость системы трекинга ошибок – симптом небрежности. Только очень большие проекты должны иметь столь большие списки ошибок, что без автоматизации никак не обойтись.

- Профессионалы не создают хаоса. Они гордятся своей квалификацией. Они поддерживают код чистым, хорошо структурированным и легко читаемым. Они следуют стандартам и общепринятым практикам. Они никогда не суетятся. Представьте, что вы в состоянии клинической смерти парите над своим телом, над которым работает кардиохирург. У хирурга очень жесткое ограничение времени – он должен успеть все сделать до того, как насос системы жизнеобеспечения не повредит вам слишком много клеток крови. Как бы вы хотели, чтобы он действовал? Хотели бы вы, чтобы он вел себя как типичный разработчик, суетясь и сея хаос? Хотели бы вы услышать «Я исправлю это в следующий раз»? Или бы вы хотели, чтобы он был сдержанным и спокойно делал свою работу как можно лучше? Вы предпочтете хаос или профессионализм?

Профессионалы ответственны. Они отвечают за свою карьеру. Они отвечают за свой код. Они отвечают за уровень своей квалификации. Они не нарушают своих принципов под давлением дедлайна. И даже наоборот – под сильным давлением они лишь еще сильнее придерживаются правил, зная, что именно так – правильно.

Автор оригинала - Uncle Bob

Простота от уменьшения

(В оригинале - Simplicity Comes from Reduction)

«Переделай это…» - сказал мой босс, нажимая и держа клавишу Delete. Я смотрел на экран со знакомым многим чувством позора, пока мой код строка за строкой отправлялся в небытие.

Мой босс не отличался красноречивостью, однако он определял плохой код с первого взгляда. И он точно знал, что с таким кодом надо делать.

Я пришел на свою нынешнюю должность студентом с кучей энергии и энтузиазма, но вообще без идей о том, как же нужно писать код. Тогда мне казалось, что любую проблему можно решить, добавив куда-нибудь еще одну переменную или дописав еще одну строку кода. И однажды вместо того, чтобы улучшаться с каждой новой версией, мой код стал громоздким, сложным и далеким от целостности.

Это естественно, что в состоянии спешки вы хотите менять код как можно меньше, даже если это минимальное изменение и окажется просто ужасным. Большинство программистов оставят плохой код, опасаясь, что его переписывание займет слишком много времени. Это может сработать для кода, очень близкого к рабочему состоянию. Но бывает код, которому уже ничего не поможет.

Для спасения плохого кода может потребоваться слишком много времени. Как только что-то становится пожирателем ресурсов, это должно быть отброшено. Быстро.

Нет, конечно, необязательно выбрасывать все написанное. Реакция моего босса была крайностью, но при этом она заставила меня хорошо подумать над следующей версией. Однако, лучшим вариантом для исправления плохого кода все равно будет безжалостный рефакторинг и удаление.

Код должен быть простым. В нем должно быть как можно меньше переменных, функций, объявлений и других языковых конструкций. Лишние строки, лишние переменные, лишнее что угодно должно быть подчищено. Удалено. То, что останется, должно быть минимумом, необходимым для того, чтобы код работал так, как запланировано. Все остальное – это только лишний шум, вносящий случайность и отвлекающий от главного потока. Скрывающий важные вещи.

И конечно же, если рефакторинг не помог, то просто удалите все и напишите его заново. Такое воссоздание с нуля часто помогает избавить код от значительной части лишнего мусора.

Автор оригинала - Paul W. Homer

Разделяйте технические и логические исключения

(В оригинале - Distinguish Business Exceptions from Technical)

В большинстве случаев есть лишь две причины, из-за которых что-то может пойти не так во время выполнения программы. Это технические проблемы, мешающие работе приложения, и бизнес-логика, не дающая нам использовать приложение неправильно. В большинстве современных языков программирования, таких как LISP, Java, Smalltalk, и C#, для обеих ситуаций используется механизм исключений. Однако эти две ситуации столь различны, что их стоит аккуратно разделять и не смешивать. Если для обеих ситуаций использовать одну и ту же иерархию исключений (не говоря уже о том, чтобы использовать одни и те же классы), то это может стать источником серьезной путаницы.

Невосстановимая техническая проблема может произойти в случае ошибки в программе. Например, вы обращаетесь к 83-му элементу массива, в котором есть лишь 17 элементов. В этом случае программа ничего не может предпринять, чтобы это исправить, и должно возникнуть исключение. Или ваш код обращается к библиотеке, вызывая функцию с неправильными параметрами, вызывая описанную выше ситуацию внутри библиотеки.

Было бы ошибкой пытаться как-то решить описанную выше ситуацию. Вместо этого мы просто позволяем исключению «всплыть» на самый верх и позволить общему обработчику исключений сделать необходимую работу, чтобы сохранить систему в стабильном состоянии – откатить транзакции, записать сообщение в лог и выдать сообщение пользователю.

Один из вариантов этой ситуации – это когда вы находитетесь на стороне библиотеки, если вызывающий нарушил контракт вызова для вашей функции – передал какую-нибудь фигню в параметрах или не выполнил требуемых пред-условий. Это практически то же самое, что и обращение к 83-му элементу массива длиной в 17 элементов – проверку должна делать вызывающая сторона, если же этой проверки нет, проблема на вызывающей стороне. И правильная реакция – сгенерировать техническое исключение.

Немного другая, но все еще техническая проблема – это невозможность продолжать работать из-за проблем в среде окружения, например, отсутствие ответа от базы данных. В этом случае вы должны предполагать, что инфраструктура уже попыталась сделать все возможное, чтобы эту проблему решить, например, повторить попытку

несколько раз, так и не получив результата. Но хоть причина и другая, для вашего кода ничего не поменялось – вы по-прежнему ничего не можете сделать для решения. Поэтому вы сообщаете о проблеме при помощи исключения, «всплывающего» наверх до общего обработчика.

В противоположность описанным выше ситуациям, бывает так, что продолжение работы программы невозможно из-за предметно-ориентированной проблемы. В этом случае мы получаем ситуацию, являющуюся исключением, т.е. необычную и нежелаемую, но не вызванную ошибкой в программе. Например, если попытаться снять со счета больше денег, чем там есть. Другими словами, эта ситуация – часть логики работы, и исключение – лишь альтернативный путь выполнения программы. Такое исключение должно корректно обрабатываться сразу же в точке вызова, непосредственно вызывающей стороной. И для таких ситуаций имеет смысл создать отдельную иерархию исключений.

Смешивание технических и логических исключений в общую иерархию размывает различия между ними и запутывает клиента по поводу контракта использования, необходимых условий перед вызовом и ситуаций, требующих реакции на них. Разделение этих двух концепций делает ситуацию более ясной и повышает шансы на то, что технические исключения будут обрабатываться общим фреймворком, а логические – непосредственно в клиентском приложении.

Автор оригинала - Dan Bergh Johnsson

Разметка кода важна!

(В оригинале - Code Layout Matters)

Когда-то очень давно я работал над системой, написанной на Cobol, в которой программисты не могли менять идентацию, если только у них не было причин для реального изменения кода. Такое ограничение было введено потому, что когда-то кто-то чего-то сломал, неправильно изменив идентацию и тем самым внеся строку в одну из специальных колонок вначале. Это правило работало даже в случае неправильной идентации, что иногда случалось. И в результате при чтении кода приходилось быть внимательным, потому что идентации нельзя было доверять.

Исследования показывают, что программисты тратят гораздо больше времени на просмотр кода, чем на собственно программирование, и поэтому оптимизировать надо вещи, связанные именно с просмотром.

- *Простота просмотра*. Людям проще всего иметь дело с повторяющимися шаблонами (это еще с тех пор, когда приходилось следить за львами в саванне), и поэтому я могу себе помочь, делая так, чтобы все, что не относится напрямую к предметной области, вся «случайная сложность», вносимая языками программирования, все это отошло на задний план, будучи стандартизировано. Если код работает именно так, как выглядит, тогда моя система восприятия поможет мне найти различия. Именно поэтому я соблюдаю конвенции о том, как располагать различные части в пределах одного модуля: константы, поля, открытые методы, закрытые методы.

- *Выразительная разметка*. Всех нас учили, что нужно выделять часть времени на то, чтобы называть переменные правильными именами, чтобы код сам объяснял то, что он делает. Разметка кода – еще одна часть этой выразительности. Первое, что надо сделать – чтобы вся команда согласилась на автоматическое основное форматирование, после чего бы использовала ручное форматирование в процессе написания. Обычно команда быстро приходит к соглашению об общем стиле «доделано вручную». Автоматическое форматирование не может понять моих намерений (я это знаю наверняка, поскольку я его писал однажды), а для меня важно, чтобы разметка отображала именно намерения, а не только синтаксис языка.

- *Компактное форматирование*. Чем больше помещается на экране, тем больше я могу видеть без разрыва контекста в виде скроллинга, что означает, что мне нужно меньше кода держать в голове. Длинные комментарии к функциям и множество

> пробелов были актуальны для восьмисимвольных имен и матричных принтеров, но сейчас есть IDE с поддержкой подсветки синтаксиса и перекрестных ссылок. Пиксели – мой сдерживающий фактор, и я хочу использовать каждый из них для понимания кода. Я хочу чтобы разметка помогала мне понимать код, но не более того.

Один мой друг-непрограммист однажды отметил, что код выглядит как поэзия. Я испытываю подобное чувство, глядя на по-настоящему хороший код, когда каждый символ в тексте имеет свое назначение и отображает стоящий за ним смысл. К сожалению, программирование не имеет такого романтичного ореола, как написание стихов.

Автор оригинала - Steve Freeman

Сделайте процесс сборки своим

(В оригинале - Own (and Refactor) the Build)

Не так и редко встречаются команды разработчиков, тщательно соблюдающие практики написания кода и при этом игнорирующие скрипты сборки, считая, что это является неважной деталью, или же считая, что это слишком сложная материя. В результате несопровождаемые скрипты сборки с повторяющимися фрагментами и множеством ошибок вызывают проблемы не реже, чем плохо написанный код.

Одна из причин, по которым многие в общем-то хорошие программисты воспринимают скрипты сборки как что-то второстепенное – это то, что они написаны на языке, отличном от основного используемого. Другая причина – то, что сборка не является в полной мере «кодом». При этом большинство программистов рады изучить новый язык, а сборка – это то, что создает выполняемую программу для конечных пользователей. Код бесполезен до тех пор, пока не будет собран. Сборка – важная часть процесса, и решения на уровне процесса сборки могут сделать код проще.

Скрипты сборки, написанные с использованием неправильных идиом, сложно сопровождаемы и, что гораздо более важно, сложно улучшаемы. Лучше потратить некоторое количество времени, чтобы понять, как это делать правильно. Ошибки могут возникнуть из-за сборки с неправильными зависимостями или проблемами в конфигурации.

По сложившейся традиции, тестирование стало задачей команды контроля качества. Тестирование также должно давать предсказуемые результаты. Примерно то же самое правило работает и для сборки – сборка должна быть в зоне ответственности команды разработки.

Понимание процесса сборки может упростить весь цикл разработки и снизить расходы. Простой скрипт сборки позволяет новому разработчику быстро влиться в поток. Автоматизация сборки позволяет получать воспроизводимые результаты у всех в команде, избегая ситуаций «А у меня оно работает». Многие инструменты сборки позволяют вам генерировать отчеты о качестве кода, давая возможность выявить проблемы на более раннем сроке. Потратив время на понимание того, как сделать скрипты сборки своими, вы можете помочь и себе, и всем остальным в команде. Вы сможете сфокусироваться на разработке функциональности, удовлетворении заказчиков и создании более приятного рабочего окружения.

Изучите все о вашем процессе сборки, чтобы быть в состоянии его менять. Скрипты сборки – это тоже код. Они слишком важны, чтобы отдавать их кому-то еще, хотя бы потому, что без них приложение не будет завершено. Работа программиста не завершается вплоть до того, когда в результате получается работающее программное обеспечение.

Автор оригинала - Steve Berczuk

Сопротивляйтесь использованию Singleton

(В оригинале - Resist the Temptation of the Singleton Pattern)

Паттерн Singleton может решить много проблем. Вы знаете, что вам необходим единственный экземпляр. Вы получаете гарантию, что этот экземпляр инициализируется перед использованием. Ваш дизайн остается простым благодаря глобальной точке доступа. Это все хорошо. Но что же не так с этим классическим паттерном?

Оказывается, проблем хватает. Опыт показывает, что большинство Singleton-ов приносят больше вреда, чем пользы. Они снижают возможность тестирования и повреждают сопровождаемость. Однако информация об этом распространена не настолько хорошо, насколько должна была бы, и в результате Singleton по-прежнему используют множество программистов. Цена же оказывается достаточно высокой:

- Очень часто необходимость в единственном экземпляре придуманная. Во множестве случаев то, что не будет нужно более чем одного экземпляра — лишь предположение. И если это делать слишком часто во множестве мест приложения — в какой-то момент могут начаться неприятности. Изменятся требования. Хороший дизайн с этим справится. А Singleton — нет.
- Singleton-ы вызывают неявную зависимость между концептуально независимыми модулями. У этой проблемы две стороны — зависимость получается скрытой, плюс добавляется связность между модулями. Проблема станет особенно острой, когда вы начнете писать юнит-тесты, предполагающие разрыв связи и возможность выборочно заменить реальный код тестовым. Singleton-ы затрудняют такую прямую подмену.
- Singleton-ы также вносят неявное состояние, что также мешает юнит-тестированию. Юнит-тесты должны быть независимы друг от друга, чтобы тесты могли запускаться в любом порядке и программа могла быть установлена в известное состояние перед выполнением каждого юнит-теста. Но как только у вас появятся Singleton-ы с изменяемым состоянием, добиться этого станет очень сложно. К тому же, такое глобально доступное состояние делает гораздо более сложным предсказание о поведении кода, особенно в многопоточных системах.
- Многопоточность добавляет Singleton-ам еще одну проблему. Поскольку прямая блокировка доступа не слишком эффективна, популярным стала так называемая блокировка с двойной проверкой. К сожалению, оказалось, что во многих языках блокировка с двойной проверкой не является потокобезопасной, и даже там, где

она все же безопасна, все еще остается возможность ее нарушить.

И наконец, еще одна проблема:

- Не существует поддержки явного удаления Singleton-а, что может стать серьезной проблемой в некоторых контекстах. Например, в архитектуре plug-in, где plug-in может быть выгружен лишь после того, как будут удалены все его объекты.
- Порядок удаления Singleton-ов при выходе из приложения неопределен. Это может привести к проблемам в некоторых приложениях, содержащих Singleton-ы со взаимосвязями. При выходе из такого приложения один Singleton может получить доступ к другому, который к тому времени уже был удален.
- Некоторые из этих проблем могут быть решены с использованием дополнительных механизмов. Однако, это приводит к дополнительной сложности кода, которую можно избежать, выбрав альтернативный дизайн.

Поэтому ограничивайте применение шаблона Singleton только к тем классам, которые действительно не должны быть созданы в более чем одном экземпляре. Не используйте Singleton в качестве глобальной точки доступа из произвольного места в коде. Прямой доступ к Singleton-у должен быть только из небольшого количества хорошо определенных мест, откуда Singleton может передаваться через интерфейсы в другие части кода. Другие части кода не будут в таком случае зависеть от того, реализован интерфейс как Singleton или нет. Такой подход разрывает зависимости, мешающие юнит-тестированию, и улучшает сопровождаемость. В следующий раз, когда вы подумаете о реализации Singleton, сделайте паузу и подумайте еще раз.

Автор оригинала - Sam Saariste

Состояние потока и парное программирование

(В оригинале - Pair Program and Feel the Flow)

Представьте, что вы полностью поглощены тем, что вы делаете – сфокусированы, вовлечены, погружены в процесс. Вы можете потерять счет времени. Вы чувствуете себя счастливыми. Вы чувствуете состояние потока. Однако достичь и оставаться в потоке для всей команды разработки слишком сложно – слишком много разнообразных мешающих факторов, легко это состояние разрушающих.

Если вы уже имели опыт программирования в парах, вы уже наверное знаете, как парное программирование способствует достижению потока. Если такого опыта у вас еще не было, мы хотим поделиться нашим, чтобы смотивировать вас попробовать это прямо сейчас. Для успешного программирования в парах усилия должны приложить как каждый член команды, так и команда целиком.

Как член команды, будьте терпеливы с менее опытными разработчиками. Не бойтесь более продвинутых разработчиков. Поймите, что все люди разные, и цените это. Знайте о своих сильных и слабых сторонах, а также о сильных и слабых сторонах других членов команды. Вы удивитесь тому, насколько многому вы можете научиться у ваших коллег.

На уровне команды продвигайте парное программирование для распределения знаний и опыта. Вы должны решать задачи в парах и менять пары и задачи достаточно часто. Договоритесь о правилах замены пар. Уточняйте их по мере надобности. Наш опыт показывает, что не обязательно завершать задачу до замены пар. И хотя передача задачи, сделанной наполовину, другой паре может выглядеть противоестественным, но наш опыт показал, что это вполне работает.

Вот несколько ситуаций, когда состояние потока разрушается, а парное программирование помогает его сохранить:

- **Снижение «фактора грузовика»**. Немного цинично, но все же: скольких членов вашей команды должен переехать грузовик, чтобы команда не смогла успешно завершить проект? Другими словами, насколько вы зависите от конкретного разработчика? Знания распределены или сосредоточены? Если вы используете ротацию пар и задач, то всегда будет кто-то еще, обладающий такими же знаниями и способный завершить задачу. В результате поток команды менее подвержен «фактору грузовика».

- **Эффективное решение проблем**. Если вы программируете в парах и сталкиваетесь с проблемой, то у вас всегда есть с кем ее обсудить. Подобные обсуждения наиболее вероятно приведут к результату, а в одиночку вы могли бы застрять. А из-за ротации задач ваше решение станет известно следующей паре, которая может его улучшить, если вы выбрали не самый оптимальный вариант.
- **Устойчивость к прерываниям**. Если кому-то нужно будет задать вам срочный вопрос, или зазвонит телефон, или вам нужно будет ответить на срочный е-мейл, то ваш партнер в паре может продолжать двигаться вперед. Когда вы закончите свои дела и вернетесь к проекту, ваш партнер будет все еще в потоке и вы быстро присоединитесь к нему.
- **Быстрое включение новичков**. В парном программировании, особенно при правильной ротации пар и задач, новички быстро входят в курс дела, узнают особенности кода и других членов команды.

Состояние потока невероятно продуктивно, но при этом и легко нарушаемо. Сделайте все, чтобы в него войти, и оставайтесь в нем, когда вам удалось его поймать.

Авторы оригинала - Gudny Hauknes, Ann Katrin Gagnat, и Kari Røssland

Тестеры - лучшие друзья программистов

(В оригинале - News of the Weird: Testers Are Your Friends)

Хотя они сами называют себя Quality Assurance, очень часто программисты называют их источником проблем. Мой опыт показывает, что очень часто между программистами и тестерами наблюдаются отношения противостояния. «Они слишком требовательны» и «Они хотят совершенства везде» - наиболее частые претензии к тестерам. Знакомо?

Я не знаю точно, почему, но у меня полностью противоположное отношение к тестерам. Возможно, потому что моим первым тестером на моей первой работе была секретарша компании. Маргарет была приятной женщиной, поддерживающей работу всего офиса и старавшейся обучить пару молодых программистов правилам поведения с заказчиками. У нее также был дар находить самые невероятные ошибки в программах.

Тогда мне пришлось работать над программой, написанной бухгалтером, считавшим себя программистом. Вы уже поняли, что проблем там было очень много. И каждый раз, когда я думал, что я что-то исправил, Маргарет пыталась это использовать, и практически всегда у нее все рушилось – каждый раз она находила новое сочетание клавиш и действий для этого. Иногда это раздражало, но она была столь приятным человеком, что я никогда не осуждала ее за то, что она выставляет меня с не самой лучшей стороны. И вот настал день, когда Маргарет смогла запустить программу, ввести платежку, распечатать и выйти из нее. Более того, когда мы начали устанавливать программу заказчикам, она всегда работала и там! У них не возникало никаких проблем, потому что Маргарет помогла мне найти их все и исправить.

Вот почему тестеры – это ваши друзья. Вам может показаться, что тестеры выставляют вас в не очень приглядном свете, заявляя порой о банальных ошибках. Но когда клиенты в восторге от того, что им не мешают все те «мелочи», которые для вас обнаружил ваш контроль качества, то вы чувствуете себя молодцом. Вы поняли идею?

Представьте: вы тестируете утилиту, использующую «невероятный алгоритм искусственного интеллекта» для поиска проблем мультизадачности. Вы запускаете ее и краем глаза замечаете, что на заставке в слове «интеллекта» - опечатка. Упс, сразу же где-то глубоко в вас зарождаются сомнения. Хотя это всего лишь опечатка, не так ли? А потом вы обнаруживаете, что там, где должны быть радио-кнопки, почему-то стоят чекбоксы, а указанные сочетания клавиш не работают так, как указано. Ни один из этих багов не является чем-то критичным, но по мере их накопления вы все больше

и больше сомневаетесь в качестве этого ПО. Если они не могут обнаружить и исправить такие мелочи, то что уж говорить про искусственный интеллект и поиск сложных проблем в многозадачных системах?

Да, они могут быть гениями, столь увлеченными искусственным интеллектом, что на такие мелочи просто не обращают внимания. И без требовательных тестеров, способных указать на такие проблемы, вы сталкиваетесь с этими проблемами в ходе работы, а в результате начинаете сомневаться в компетентности программистов.

Поэтому как бы странно это не звучало, именно те тестеры, которые находят каждую мелочь в вашем коде, являются вашими друзьями.

Автор оригинала - Burk Hufnagel

Тестирование - обязательный этап разработки

(В оригинале - Testing Is the Engineering Rigor of Software Development)

Программисты любят использовать метафоры, объясняя, что они делают, членам семьи и другим нетехническим людям. Часто при этом происходит сравнение со строительством моста и другими «реальными» примерами проектирования. Однако эти метафоры перестают работать, если попытаться зайти в их использовании слишком далеко. Все же разработка ПО сильно отличается от строительства материальных объектов.

Если применить аналогии разработки ПО к строительству моста, то строители моста должны после его постройки провезти по нему что-нибудь тяжелое. Если мост выстоит – значит, он построен правильно. Если же нет – значит, надо искать и исправлять проблемы в чертеже и начинать все заново. Все последнее тысячелетие инженеры разрабатывали математический аппарат, позволяющий им просчитать конструкцию материального предмета до его реальной постройки и без его реальной постройки. В разработке ПО такого аппарата пока нет и, похоже, не предвидится именно потому, что разработка ПО сильно отличается от реального строительства. Очень детальное сравнение сделал Джек Ривз (Jack Reeves) в статье «Что такое дизайн ПО» ("What is Software Design?") в журнале «C++» (C++ Journal) в 1992 году. Несмотря на то, что написано это было почти двадцать лет назад, изложенный там материал все еще актуален. Он нарисовал достаточно мрачную картину, но тогда, в 1992-м году еще не была достаточно развита традиция интенсивного тестирования ПО.

Тестирование в материальном мире – трудная задача, ведь вам необходимо физически построить то, что вы собираетесь тестировать. Построить что-то лишь для того, чтобы посмотреть, что получится в результате – звучит как-то не очень оптимистично. Однако «стройка» в программировании обходится значительно дешевле. И специально для тестирования разработаны целые системы инструментов – юнит-тестирование, мок-объекты (mock-objects), платформы для тестирования и другие подобные вещи. Инженеры материального мира могут только мечтать о том, чтобы что-нибудь построить для тестирования в реальных условиях. Как разработчики ПО, мы должны принять тестирование как основное (но не единственное) средство проверки работоспособности ПО. Вместо ожидания появления какого-нибудь механизма, позволяющего просчитать поведение ПО (по аналогии с материальным миром), мы должны использовать то, что у нас уже есть. Посмотрев на процесс с этой точки зрения, мы теперь можем адекватно возражать менеджеру, говорящему «У нас

нет времени на тестирование». Ведь строитель моста никогда не услышит от своего босса «Никакого структурного анализа – нас поджимает дедлайн!». Понимание того, что тестирование – это путь к качественному ПО, позволяет нам, разработчикам, отбрасывать аргументы против тестирования как непрофессиональные.

Да, тестирование требует времени. Также как и структурный анализ материального объекта требует времени. Оба процесса обеспечивают качество конечного продукта. Пришло время программистам взять ответственность за производимый ими продукт. Одного тестирования для этого недостаточно, но оно необходимо. Тестирование – это обязательная часть процесса разработки.

Автор оригинала - Neal Ford

Тестируйте по ночам и в выходные

(В оригинале - Test While You Sleep (and over Weekends))

Расслабьтесь. Я не имею в виду удаленный центр разработки, выход на работу в выходные или введение ночных смен. Вместо этого я хочу обратить внимание на то, сколько вычислительной мощности имеется в нашем распоряжении. Особенно о том, насколько мы не используем его для того, чтобы облегчить себе жизнь. Вам все время не хватает вычислительной мощности в течение дня? Если это так, то что делают ваши тестовые сервера в нерабочее время? Чаще всего, они простаивают, как по ночам, так и по выходным. И вы можете это использовать.

- *Приходилось ли вам помещать изменение в систему контроля версий без выполнения полного набора тестов?* Одной из причин, почему программисты не запускают тесты перед помещением, является долгое время выполнение полного набора тестов. Когда дедлайн поджимает, люди начинают искать короткие пути. Одно из решений – разбить тесты на две группы. Маленькая, при этом обязательная группа тестов, быстро выполняющаяся, поможет убедиться в том, что перед каждым коммитом проект все еще как-то работает. И большая (при этом включающая в себя и первую маленькую группу тоже, на всякий случай), запускающаяся ночью автоматически и выдающая полный отчет к утру следующего дня.
- *У вас достаточно возможностей протестировать стабильность вашего продукта?* Долгоисполняющиеся тесты важны для обнаружения утечек памяти и других проблем стабильности. Они редко запускаются в рабочее время из-за ограниченности ресурсов. Вы можете запускать эти тесты в пятницу вечером, и до утра в понедельник тесты смогут проработать 60 часов без перерыва.
- *Вам достаточно времени для проведения тестов производительности?* Я видел множество команд, сражающихся за доступ к оборудованию для тестирования производительности. В большинстве случаев командам не хватало времени для работы на нем в течении дня, при этом оборудование простаивало в остальное время. И сервера, и сеть менее всего нагружены по ночам и выходным, и в это время лучше всего проводить тесты производительности.
- *Слишком много комбинаций для тестирования?* Во многих случаях продукт должен работать на нескольких платформах, например, на 32 и 64-х битных Windows, Linux и Solaris, или же на различных версиях одной операционной системы. При этом тестируемое приложение может использовать множество различных транспортных механизмов и протоколов (HTTP, AMQP, SOAP, CORBA и т.п.). Тестирование всех комбинаций вручную очень затратно по времени и чаще

> всего делается ближе к дате выпуска из-за нехватки ресурсов. И это может быть слишком поздно. Автоматизированные тесты, запускаемые по ночам или на выходных, могут протестировать все комбинации более тщательно.

Немного подумав и изучив скрипты, вы можете настроить несколько планировщиков для запуска тестов по ночам и выходным. Также в наличии имеются специальные инструменты для тестирования, которые могут в этом помочь. Некоторые организации устраивают матрицы серверов, объединяющие сервера различных отделов и команд в общий пул для более эффективного их использования. В таком случае вы можете отправлять тесты на выполнение ночью или на выходных.

Автор оригинала - Rajith Attapattu

Тестируйте требуемое поведение, а не случайное

(В оригинале - Test for Required Behavior, not Incidental Behavior)

Общая проблема тестирования – предположить, что вам нужно тестировать точно то, что делает приложение. На первый взгляд это предположение звучит вполне здраво. Но если озвучить его немного по-другому, то проблема будет более заметной: проблема тестирования – привязать тесты к специфике реализации, когда эта специфика случайна и не относится к желаемой функциональности.

В случае такой привязки к деталям реализации при изменениях в реализации, совместимых с требованиями к поведению системы, тесты могут провалиться, давая ложноположительный результат. Программисты обычно в этом случае либо переписывают тесты, либо меняют код. Предположение, что ложноположительное срабатывание указывает на реальную ошибку, может приводить к страху, неуверенности и беспокойству. Возрастает ощущение того, что программа ведет себя нестабильно. В случае переписывания теста программисты либо переделывают его так, чтобы он затрагивал желаемую функциональность (это правильно!) или же приспосабливают тест к новой реализации (а так делать не стоит!). Тесты должны быть точными, но при этом должны быть и аккуратными.

Например, сравнение вроде `strcmp` в C или `String.compareTo` в Java требует, чтобы результат был отрицательным, если левая часть меньше правой, положительным, если наоборот и нулевым, если обе части равны. Этот стиль сравнения используется множеством API, включая `comparator` для функции `qsort` в C и `CompareTo` в `Comparable` интерфейсе в Java. И хотя в реализациях очень часто используются значения `+1` и `-1`, считать, что эти значения являются требованием к реализации, будет ошибкой. Написание теста исходя из этого предположения приведет к тому, что ошибочное предположение материализуется.

Подобные вещи происходят с тестами, проверяющими количество пробелов, точное написание слов и другие аспекты форматирования, часто являющиеся случайными. Кроме случаев, когда вы пишете XML-генератор, предлагающий настраиваемое форматирование, количество пробелов не должно быть значимым для итогового результата. Точно также жесткая привязка к позициям элементов GUI снижает возможность дальнейшего изменения. Небольшое и незначимое изменение в реализации или форматировании в результате внезапно становится «убийцей» сборки.

Слишком специфические тесты часто являются проблемой при тестировании по принципу «белого ящика». В этом случае структура кода определяет необходимые тест-кейсы. Типичная ошибка при этом – что тестирование заключается в проверке того, что код делает именно то, что он делает. Такие тесты не добавляют ценности и лишь приводят к ложной уверенности в прогрессе и стабильности.

Чтобы быть эффективными, тесты должны проверять только контрактные обязательства, а не реализацию. Тестируемый юнит должен рассматриваться как «черный ящик», предоставляющий интерфейс в исполняемом виде. Поэтому старайтесь, чтобы тестируемое поведение было идентичным требуемому.

Автор оригинала - Kevlin Henney

Тесты должны быть точными

(В оригинале - Test Precisely and Concretely)

Важно тестировать желаемое, требуемое поведение кода, а не случайное поведение конкретной реализации. Однако это не должно быть оправданием для некорректно работающих тестов. Тесты должны быть точными.

Как обычно, сортировка – очень показательный пример. Реализация алгоритмов сортировки – не очень частая задача для программиста, однако сортировка – настолько простое понятие, что большинство людей верят, что они знают, чего от нее ожидать. Эта кажущаяся простота, однако, делает труднообнаруживаемыми неправильные предположения.

Если у программиста спросить: «А что же именно вы будете тестировать», наиболее частым ответом будет: «Результат сортировки – отсортированная последовательность элементов». И хотя это действительно так, это еще не все. Если попросить уточнить критерий, многие добавят, что результирующая последовательность должна быть той же длины, что и исходная. И это так, но этого все еще недостаточно. Возьмем для примера вот такую последовательность:

```
3 1 4 1 5 9
```

Следующая последовательность удовлетворяет критерию отсортированности и такой же длины, как и у входящей последовательности:

```
3 3 3 3 3 3
```

И хотя она и удовлетворяет критерию, очевидно, что она явно не то, что ожидается от алгоритма сортировки. Пример этот взят из реального кода (к счастью, проблема была «поймана» буквально перед самым выпуском), в котором из-за примитивной ошибки первый элемент входящей последовательности «размножался» на всю ее длину.

Полный критерий выглядит так: результат – отсортированная последовательность, при этом являющаяся перестановкой исходной последовательности. Это точно ограничивает требуемое поведение. Требование одинаковой длины отпадает как ненужное.

Но даже правильного вышеприведенного критерия еще недостаточно, чтобы написать хороший тест. Хороший тест должен быть легко читаемым. Он должен быть всеобъемлющим и при этом простым, чтобы было видно, правилен ли он. И если только у вас уже нет готового отлаженного кода, проверяющего, что последовательность отсортирована и что две последовательности являются перестановкой друг друга, то скорее всего, тестирующий код будет гораздо сложнее кода тестируемого. Как заметил Тони Хор (Tony Hoare):

> Есть два варианта проектирования ПО: или сделать все настолько просто, что проблем не будет, или сделать все настолько сложно, что проблем не будет видно.

Использование конкретного случая устраняет излишнюю сложность и возможность для появления проблем. Например, для входной последовательности:

```
3 1 4 1 5 9
```

результат сортировки должен быть

```
1 1 3 4 5 9
```

Любой другой результат является неприемлемым.

Конкретные примеры помогают проиллюстрировать общее поведение простым и однозначным способом. Результат добавления элемента в пустое множество – не просто факт, что множество теперь не пустое, а то, что в множестве сейчас единственный элемент, и этот элемент совпадает с добавленным. Поскольку не пустое множество – это и два, и более элементов, что, однако, будет неправильным результатом. Результат добавления строки в таблицу не просто увеличение количества строк в таблице на один, а и то, что эту строку можно из таблицы извлечь по ее ключевому полю. И так далее.

При определении поведения тесты должны быть не только правильными, но и точными.

Автор оригинала - Kevlin Henney

Только код расскажет всю правду

(В оригинале - Only the Code Tells the Truth)

Поведение программы полностью определяется ее исполняемым кодом. Если у вас есть только исполняемый файл, разобраться в коде будет непросто. Однако исходный код должен быть доступен, если программа ваша, или же если вы имеете дело с open source или с купленной коммерческой разработкой, или же код написан на интерпретируемом языке. Взгляда на исходный код должно быть достаточно для понимания того, что делает программа. Более того, только исходный код даст вам все ответы. Даже наиболее детально проработанные исходные требования не сообщат вам всего, они содержат только высокоуровневые намерения. Дизайн содержит больше деталей, но и он все равно еще не содержит всех деталей реализации. К тому же эти документы могут не полностью соответствовать реализации. Или вообще могут не быть написаны. И исходный код – последнее, что у вас остается.

А теперь спросите себя, насколько ясно ваш код сообщает вам или кому-нибудь еще о том, что именно он делает?

Вы можете подумать, что «Я напишу комментарии, которые все объяснят». Но помните, что комментарии не исполняются. Они точно так же могут не соответствовать действительности, как и любой другой документ. По традиции принято считать, что комментарии – это хорошо по определению, и многие программисты пишут их все больше и больше, даже если комментарии и повторяют объяснения тривиальных вещей, и так ясных из кода. Этот путь – неправильный. Если вашему коду требуются комментарии, перепишите его так, чтобы комментарии не были бы нужны вообще. Объемные комментарии мешают, и некоторые IDE могут их даже самостоятельно скрывать. Если вам нужно объяснить причину изменения, добавьте комментарий в систему контроля версий, но не добавляйте его непосредственно в код.

А что же делать для того, чтобы код был как можно более ясным и понятным? Выбирайте правильные имена. Структурируйте код с учетом функциональности. Старайтесь сделать его без повторений. Пишите автоматизированные тесты, объясняющие предполагаемое поведение, и проверяйте использование интерфейсов. Переписывайте код сразу же, как только видите более простое и ясное решение. Делайте код как можно более простым для понимания.

Воспринимайте код как литературное произведение – поэму, эссе, дневник или важное письмо. Старайтесь писать выразительно, так, чтобы код делал то, что должен и при этом как можно более ясно сообщал, что он делает, так, чтобы донести ваши

намерения до читателя, когда вас не будет рядом. Помните, что легко используемый код используется гораздо дольше, чем это может предполагаться. Те, кому придется его сопровождать, не раз вас поблагодарят. А если вы как раз занимаетесь поддержкой кода, который написан так, чтобы затруднить вам работу, то смело применяйте изложенные выше принципы, чтобы его улучшить. Приведите код в порядок и старайтесь его поддерживать.

Автор оригинала - Peter Sommerlad

Убунту-программирование

(В оригинале - Ubuntu Coding for Your Friends)

Часто мы пишем код в изоляции, и код отражает нашу личную интерпретацию проблемы и наше видение решения. Мы можем при этом работать в команде, все еще оставаясь изолированной командой. Мы слишком часто забываем, что этот код, написанный в изоляции, будет запускаться, использоваться, меняться другими людьми. Очень легко не заметить социальную сторону написания ПО. Создание ПО – это смесь технической и социальной работы. Нам нужно чаще поднимать голову, чтобы понять, что мы не работаем в изоляции и на всех нас лежит часть ответственности за повышение вероятности успеха для всех, а не только для команды разработки.

Вы можете писать качественный код в изоляции, замкнувшись в себе. Это так называемая эгоцентрическая позиция. Этого же придерживается Дзен, и это все про вас в тот момент, когда вы пишете код. Я всегда стараюсь жить в настоящем моменте, потому что это помогает мне писать качественно, но при этом я живу в своем собственном моменте. А как же момент команды? Он тот же самый, что и мой, или нет?

На зулусском языке, философия Убунту формулируется как «Умунту нгумунту нгабанту», что можно приблизительно перевести как «Личность – это личность среди личностей». Я становлюсь лучше, потому что вы делаете меня лучше своими хорошими поступками. Обратная сторона – вы начинаете делать хуже свою работу, если я делаю хуже свою. Для разработчиков это можно сузить до «Разработчик – это разработчик среди других разработчиков». А опустившись до материального мира, можно сказать «Код – это код среди другого кода».

Качество моего кода влияет на качество вашего. Что, если мой код – плохой? Даже если вы напишете очень хороший код, то будут точки соприкосновения вашего кода с моим, что снизит качество вашего. Вы можете применять различные паттерны и техники для снижения ущерба, но ущерб все равно уже нанесен. Ведь я заставил вас делать больше, чем вам было нужно лишь потому, что я не подумал о вас когда проживал свой момент.

Я могу считать свой код аккуратным, но я все еще могу его улучшить при помощи Убунту-программирования. Как выглядит Убунту-код? Он выглядит просто как хороший код. Речь не идет о самом написанном коде. Речь идет о процессе его написания.

Написание кода для ваших друзей, согласно философии Убунту, поможет команде разделить ваши ценности и принципы. Следующий, кто каким-либо образом коснется вашего кода, будет лучше и как человек, и как программист.

Дзен – это про отдельную личность. Убунту – это Дзен для группы людей. Мы очень-очень редко пишем код лишь для себя самих.

Автор оригинала - Aslam Khan

Удобство?

(В оригинале Convenience Is not an -ility)

О важности проектирования хорошего API уже было сказано очень много. Хороший API трудно сделать с первого раза и еще труднее изменить его потом. Большинство программистов выучили, что хороший API соответствует уровням абстракции, демонстрирует логичность и симметрию, а также формирует словарь выразительного языка. Увы, одно знание о принципах не приводит к желаемому результату.

Вместо чтения возвышенных проповедей я хочу лишь обратить внимание на одну стратегию API, которую я открываю снова и снова – аргумент удобства. Обычно все начинается с догадки вроде:

- Я не хочу, чтобы другим классам приходилось делать два вызова для того, чтобы сделать вот это.
- Зачем мне нужна еще одна функция, если эта уже делает почти то же самое. Я лучше добавлю туда еще один параметр-переключатель.
- Смотрите, все просто. Если второй параметр заканчивается на .txt, то функция автоматически решит, что первый параметр – это имя файла, и вторую функцию можно не делать.

Однако несмотря на благие намерения, это приводит к снижению читабельности кода, использующего такой API. Вызов такого метода:

```
parser.processNodes(text, false);
```

практически бессмысленен без знания подробностей реализации или чтения документации. И скорее всего, метод был написан для удобства его писавшего в противоположность удобству использующего. На самом деле фразу «Я не хочу, чтобы надо было вызывать два метода» следует читать как «Я не хотел писать два отдельных метода». Ничего радикально неправильного в удобстве нет, если говорить об удобстве как о противоядии к монотонности, неуклюжести или неповоротливости. Однако, если чуть глубже копнуть, то противоядием к перечисленным симптомам будет эффективность, логичность и элегантность, а вовсе не удобство. API предназначен для скрытия нижележащей сложности, и вполне логично ожидать, что проектирование хорошего API потребует определенных усилий. Написать один большой метод будет гораздо удобнее, чем продумать адекватный набор операций, но вот будет ли этот метод более удобен в использовании?

Сравнение API с языком может привести нас к лучшему пониманию того, какое решение следует принять. API должен предоставить словарь, дающий возможность вышележащему уровню задавать нужные вопросы и получать ответы. Необязательно, чтобы для каждого вопроса использовалось единственное слово. Разнообразие словаря может позволять использовать различия в значениях. Так, мы скорее предпочтем метод run вместо метода walk(true), даже если они будут полностью эквивалентными. Последовательный и хорошо продуманный словарь, используемый для API, сделает код следующего уровня более выразительным и понятным. Еще более важно, что если словарь можно будет комбинировать, то другие программисты смогут использовать API так, как вы и не подозревали – на самом деле отличное удобство для использования. Когда в следующий раз захотите объединить несколько вещей в один метод, вспомните, что в языке нет слов вроде УбратьВКомнатеИСделатьУроки, даже если вам такое слово кажется удобным для обозначения часто используемого действия.

Автор оригинала - Gregor Hohpe

Удовлетворяйте свои амбиции на проектах open source

(В оригинале - Fulfill Your Ambitions with Open Source)

С высокой долей вероятности вы разрабатываете совсем не те программы, которые бы вам хотелось разрабатывать. Например, вы можете работать на проекте для крупной страховой компании, а хотеть при этом работать где-нибудь в Google, Apple, Microsoft или начать свой собственный стартап. Однако вы не сможете устроиться на работу туда, куда вы хотите, не имея опыта разработки в этой области.

К счастью, у этой проблемы есть решение: open source. Существуют тысячи проектов с открытым исходным кодом в самых разнообразных областях. Если вам нравится идея разрабатывать операционные системы, выбирайте любой из дюжины доступных проектов операционных систем. Если вы хотите работать над ПО обработки звука, анимации, криптографии, игр, как индивидуальных, так и онлайновых много пользовательских, или любым другим типом ПО, вы практически наверняка найдете как минимум один проект с открытым кодом, посвященный именно этой теме.

Разумеется, за все придется платить. В данном случае платить вы будете своим свободным временем. Вы ведь не можете разрабатывать open source проект в рабочее время – вы ведь должны выполнять свои прямые обязанности. А материальное вознаграждение за участие в open source проекте получают очень мало людей. У вас должно быть желание потратить часть своего личного времени. Чем больше вы будете работать над open source проектом, тем быстрее вы сможете реализовать ваши настоящие амбиции в программировании. При этом важно учитывать юридические детали вашего контракта с работодателем – там может быть пункт, запрещающий вам писать ПО определенной тематики даже в свободное время. А также вам нужно быть осторожным, чтобы нигде не нарушить закон в связи с авторскими правами, патентами, торговыми марками и промышленными секретами.

Open source предоставляет неограниченные возможности для мотивированного программиста. Во-первых, вы начнете общаться с теми, кто уже занимается тем, что вам интересно, и сможете многому научиться, изучая их код. Во-вторых, вы сами начнете помещать свои идеи и код в проект. Не каждая ваша идея будет принята, но какие-то из них будут. А вы получите новый опыт, работая над проблемами проекта. И в-третьих, вы встретите людей с теми же самыми интересами, что и у вас – такая open

source дружба может длиться всю жизнь. И в-четвертых, когда вы станете компетентным участником проекта, вы сможете внести свой вклад в технологию, для вас интересную.

Начать работать с open source проектом достаточно несложно. Документация по необходимым инструментам (системы контроля версий, редакторы, языки программирования, компиляторы) обычно присутствует в изобилии. Найдите интересный для вас проект и изучите используемый в нем инструментарий. Документации непосредственно о самом проекте обычно крайне мало, но это не играет большой роли. Ведь лучший способ чему-то научиться – это изучить код самостоятельно. Если вы хотите присоединиться к проекту, вы можете предложить помощь в составлении документации. Или в написании тестовых сценариев. И хотя это может звучать не слишком захватывающе, правда в том, что написание тестовых сценариев для чьего-то чужого кода – один из самых быстрых способов обучиться чему-то новому. Просто пишите тестовые сценарии. Хорошие тестовые сценарии. Находите ошибки. Предлагайте исправления. Ищите друзей. Работайте над интересным для вас проектом. И удовлетворяйте свои амбиции.

Автор оригинала - Richard Monson-Haefel

Упущеные возможности полиморфизма

(В оригинале - Missing Opportunities for Polymorphism)

Полиморфизм – одна из фундаментальных идей объектно-ориентированного подхода. Перевод этого слова с греческого – «множество форм». В контексте программирования речь идет о множестве форм конкретного класса, объекта или метода. Но полиморфизм – это не только альтернативные реализации. Будучи правильно использованным, полиморфизм позволяет обходиться без конструкций if-then-else. Нахождение внутри контекста позволяет нам выполнить нужный код непосредственно, тогда как нахождение вне контекста вынуждает переделать код так, чтобы сделать это правильно. Аккуратно используя альтернативные реализации, мы сможем сделать код более компактным, а значит, более сопровождаемым. Лучше всего это продемонстрировать на примере, скажем, упрощенной корзины покупок:

```
public class ShoppingCart {
    private ArrayList<Item> cart = new ArrayList<Item>();
    public void add(Item item) { cart.add(item); }
    public Item takeNext() { return cart.remove(0);  }
    public boolean isEmpty() { return cart.isEmpty(); }
}
```

Пусть наш интернет-магазин предлагает два вида товаров – которые можно скачать, и которые надо посылать почтой. Создадим объект, который поддерживает следующие операции:

```
public class Shipping {
    public boolean ship(Item item, SurfaceAddress address) { ... }
    public boolean ship(Item item, EMailAddress address { ... }
}
```

После завершения покупки необходимо выполнить доставку:

```
while (!cart.isEmpty()) {
    shipping.ship(cart.takeNext(), ???);
}
```

«???» здесь вовсе не какой-то новый супероператор, а всего лишь вопрос, ответ на который указывает, надо ли послать этот товар по электронной почте или же по обычной. Однако контекст, необходимый для ответа на этот вопрос, уже не существует. Мы могли бы использовать флаг или enum для этого и применить конструкцию if-then-

else. Другое же решение – создать два класса, расширяющих функциональность класса item. Назовем их для примера DownloadableItem и SurfaceItem. Теперь напишем код. Сделаем Item интерфейсом с единственным методом ship. Чтобы выполнить доставку, нужно будет вызвать item.ship(). Реализация метода ship будет в классах DownloadableItem и SurfaceItem.

```
public class DownloadableItem implements Item {
    public boolean ship(Shipping shipper) {
        shipper.ship(this, customer.getEmailAddress());
    }
}

public class SurfaceItem implements Item {
    public boolean ship(Shipping shipper) {
        shipper.ship(this, customer.getSurfaceAddress());
    }
}
```

В этом примере мы делегировали ответственность за правильную доставку объекту Item. Поскольку каждый объект Item знает как его нужно доставить, это позволяет нам обойтись без конструкции if-then-else. Данный код также демонстрирует использование двух паттернов проектирования, часто работающих в паре: Command и Double Dispatch. Эффективное использование этих паттернов основано на правильном использовании полиморфизма. Такое использование приведет к уменьшению количества конструкций if-then-else в коде.

И хотя бывают случаи, когда использование if-then-else оправдано более, чем применение полиморфизма, чаще всего полиморфизм дает в результате более компактный и более ясный для понимания код. Количество упущенных возможностей – это просто количество конструкций if-then-else в вашем коде.

Автор оригинала - Kirk Pepperdine

Установи меня

(В оригинале - Install Me)

Я не просто зашел поинтересоваться вашей программой.

У меня сейчас туева хуча проблем и длиннющий список того, что надо сделать. И единственная причина, почему я на вашем веб-сайте – это то, что я где-то слышал, что ваша программа может решить все мои проблемы. Поэтому думаю вы простите мне мой скептицизм.

Если исследования о работе зрительной системы не врут, то я уже прочитал заголовок и ищу синие подчеркнутые слова *«Скачать прямо сейчас»*. Да, кстати, если я пользуюсь Линуксовым браузером и у меня IP из Великобритании, то скорее всего мне понадобится версия под Линукс с европейского зеркала. И не надо меня об этом спрашивать. Предполагая, что после клика сразу же появится окно загрузки, я указываю, куда сохранить файл и продолжаю чтение.

Все мы непрерывно анализируем соотношение затрат и выгод. И если ваш проект хотя бы на секунду покажется мне несоответствующим моим ожиданиям, я просто закрою его страницу и пойду поищу что-нибудь еще. Мгновенное удовольствие превыше всего.

Первый барьер – это инсталляция. Вам не кажется этой проблемой? Тогда зайдите в свою папку *Download* и посмотрите, что там лежит. Множество архивов, не так ли? И сколько из этого вы реально используете? У меня лишь треть делает что-то еще, кроме занимания места на диске.

Мне не нравится, когда кто-то входит в мой дом без приглашения. Перед запуском инсталляции я хочу точно знать, куда будет установлено ваше приложение. Это ведь мой компьютер и я хочу контролировать в нем все. Я также хочу иметь возможность в любой момент удалить ваше приложение. И если я заподозрю, что у меня это не получится, я не буду его устанавливать вообще. Мой компьютер в настоящий момент стабилен и я хочу, чтобы он таким и оставался.

Если ваше приложение имеет графический интерфейс, я хочу сделать что-нибудь простое и сразу увидеть результат. Wisard-ы тут не помощники, поскольку я не понимаю, как они работают. Скорее всего, я захочу открыть или отредактировать какой-нибудь файл. Я вряд ли захочу создать проект, импортировать папки или сообщить свой е-мейл. Если до сих пор все работает, тогда я загляну в tutorial.

Если ваше ПО – это библиотека, то я продолжу изучать ваш сайт в поисках инструкции о быстром старте. Я хочу также аналог “Hello world” из пяти строчек и без лишнего мозготраха и с подробным описанием того, что должно получиться, если его запустить. Никаких огромных XML файлов или шаблонов, а всего лишь маленький скрипт. И да, конечно же я уже загрузил себе фреймворк вашего конкурента, того самого, который на всех форумах пишет, что его приложение на порядок лучше вашего. И если до сих пор все работает, тогда я загляну в tutorial.

Упс, а что, у вас нет tutorial-а? На том языке, который я понимаю?

И если в toturial-е будут упомянуты мои проблемы, я сдамся. Я начну читать о том, что я могу сделать при помощи вашей программы, и делать это с удовольствием. Я наконец-то выдохну, откинусь на спинку стула и сделаю глоток чая (я ведь сказал, что я из Великобритании?), а потом поиграю с вашими примерами, чтобы понять, как работать с вашим приложением. И если я решу свои проблемы с его помощью, я пошлю вам благодарственный е-мейл. Я отправлю отчет об ошибке, если таковая случится, а также напишу о желаемых дополнениях. И я расскажу всем своим друзьям о том, что ваше ПО лушчее, хотя я даже не попробовал аналогичное ПО ваших конкурентов. И все это лишь потому, что вы потрудились облегчить мои первые шаги.

Я все еще вас не убедил?

Автор оригинала - Marcus Baker

Учитесь оценивать

(В оригинале - Learn to Estimate)

Будучи программистами, вам нужно быть способными предоставлять руководству, коллегам и заказчикам оценку для выполняемых вами задач, чтобы они могли представить требуемое количество времени, средств, технологий и других ресурсов для достижения своих целей.

Чтобы делать правильные оценки, необходимо знать некоторые техники построения таких оценок. Однако прежде всего необходимо понять, что же представляет собой оценка и для чего она должна использоваться. Как бы странно это не звучало, но большинство разработчиков и менеджеров этого не понимают.

Часто можно услышать вот такой диалог между менеджером (ПМ) и программистом:

> *ПМ:* Ты мог бы сделать оценку, за сколько времени ты сделаешь такую-то функциональность?
>
> *Программист:* За месяц.
>
> *ПМ:* Это слишком долго! Функциональность должна быть готова через неделю!
>
> *Программист:* Меньше чем за три никак не получится!
>
> *ПМ:* Больше двух я никак не могу выделить!
>
> *Программист:* Хорошо, договорились.

Чтобы понять, что здесь не так, надо дать определения для трех вещей: оценки, цели и обязательства.

- *Оценка* – это примерный подсчет или суждение о количестве чего-либо, будь то стоимость, количество, объем или что-нибудь еще. Это определение предполагает, что оценка – это измерение, основанное на фактах: исходных данных и предыдущем опыте. Желания и хотения не должны влиять на оценку. Из определения также следует, что оценка не может быть точной. Разработчик не может оценить время выполнения задачи в 234.14 дня.
- *Цель* – это выражение желаемого на языке предметной области, например «Система должна поддерживать не менее 400 пользователей одновременно».

- *Обязательство* – это обещание доставить определенную функциональность определенного качества к определенному сроку или событию. Например, «Мы добавим поиск в следующем релизе продукта».

Оценки, цели и обязательства друг с другом не связаны, но при этом цели и обязательства должны быть основаны на озвученных оценках. Как заметил Steve McConnell, «Основная цель оценок в разработке – не предсказать дату завершения проекта, а определить, являются ли цели проекта достаточно реалистичными, чтобы проект мог их достигнуть». Таким образом, цель оценок – сделать возможным планирование и адекватное управление проектом, позволяя в результате давать конкретные обязательства всем вовлеченным сторонам.

В вышеупомянутом диалоге менеджер на самом деле спрашивал об обязательствах, основанных к тому же на несформулированных целях, существующих лишь в его голове, а вовсе не об оценках. Когда вас в следующий раз попросят предоставить оценку, убедитесь, что все понимают, о чем на самом деле идет разговор – и у вашего проекта повысятся шансы на успех. А теперь пришло время поговорить о методиках…

Автор оригинала - Giovanni Asproni

Хороший интерфейс: легко использовать правильно, сложно использовать неправильно

(В оригинале - Make Interfaces Easy to Use Correctly and Hard to Use Incorrectly)

Самая часто встречающаяся в разработке ПО задача – это определение интерфейсов. Интерфейсы могут быть на самом высоком уровне абстракции (пользовательский интерфейс), на самом низком (интерфейс функций) и на остальных уровнях между ними (интерфейс классов, интерфейс библиотек и т.п.). Независимо от того, общаетесь ли вы с пользователями для определения того, как они будут взаимодействовать с системой, или же с программистами, определяя интерфейс классов, проектирование интерфейса – важная часть работы. Если вы сделаете ее хорошо, ваш интерфейс будет удобно использовать, и он будет повышать производительность других людей. Плохо сделанный интерфейс будет служить источником раздражения и ошибок.

Свойства хорошего интерфейса:

- **Легко использовать правильно**. Люди, использующие хорошо спроектированный интерфейс, почти всегда используют его правильно, потому что так проще. В GUI они почти всегда кликают по правильной иконке, кнопке или меню, потому что это для них очевидно. В функциональном API они практически всегда передают корректные значения в корректные функции, потому что именно так делать наиболее естественно. С интерфейсами, которые легко использовать, все просто работает само собой.
- **Сложно использовать неправильно**. Хорошие интерфейсы предвидят возможные ошибки и делают так, чтобы ошибки было сложно (а лучше – если вообще невозможно) допустить. GUI может запрещать или убирать команды, не имеющие смысла в данном контексте, или же API может сразу решить проблему неправильного порядка параметров, разрешив их передачу в произвольном порядке.

Хороший способ разработать удобный в использовании интерфейс – это изучить его до его создания. Постройте модель GUI, на доске маркерами или же на столе карточками, и попробуйте поиграть с моделью до того, как она будет реализована в коде. Напишите вызовы API до того, как будут определены функции. Пройдитесь по основным примерам использования и уточните желаемое поведение интерфейса. На что можно будет нажать? Что можно будет передать? Простые в использовании интерфейсы выглядят естественно, поскольку дают возможность вам делать то, что

вам нужно делать. У вас гораздо больше шансов сделать такой интерфейс, если вы будете на него смотреть глазами пользователя. (Это, кстати, одна из сильных сторон подхода «сначала тестируй, потом кодируй»)

Чтобы сделать интерфейс, который будет сложно использовать неправильно, нужно две вещи. Первое – вы должны предугадать ошибки пользователей и найти способы их предотвратить. Второе – вы должны отслеживать случаи неправильного использования предыдущих версий интерфейса и менять его (да-да, менять интерфейс!) так, чтобы предотвращать подобные ошибки в дальнейшем. Лучший вариант предотвращения неправильного использования – сделать его невозможным. Если пользователь настойчиво пытается отменить неотменяемую операцию, постарайтесь сделать ее отменяемой. Если пользователь часто передает неправильный параметр – постарайтесь сделать так, чтобы вариант пользователя имел смысл и мог быть корректно принят.

И главное – помните, что интерфейсы нужны для удобства тех, кто эти интерфейсы использует, а не тех, кто их разрабатывает.

Автор оригинала - Scott Meyers

Числа с плавающей точкой - не действительные!

(В оригинале - Floating-point Numbers Aren't Real)

Числа с плавающей точкой не являются действительными числами в математическом смысле, хотя и называются «Real» в некоторых языках программирования. Действительные числа обладают бесконечной точностью и поэтому непрерывные. Числа с плавающей точкой обладают ограниченной точностью и скорее похожи на «плохо работающие» целые числа, поскольку они даже не занимают весь диапазон, отведенный под целые числа.

Для иллюстрации этого возьмите и присвойте целое число 2147483647 (это максимальное 32-битное целое) 32-х битной же переменной с плавающей точкой, скажем, x. Если вы напечатаете ее, то вы увидите 2147483648. А теперь напечатайте `x - 64`. Напечатается опять 2147483648. А теперь попробуйте напечатать `x - 65`, и получите 2147483520! Почему? Потому что расстояние между двумя соседними числами с плавающей точкой – 128, и результат округляется до ближайшего существующего числа.

По IEEE числа с плавающей точкой – это числа фиксированной длины, записываемые следующим образом: $11.d_1d_2...d_{p-1} \times 2^e$, где p – это точность (24 для float, 53 для double). Расстояние между двумя ближайшими числами при этом - 2^{1-p+e}, что можно приблизительно выразить как $\varepsilon|x|$, где ε – машинный эпсилон (2^{1-p})

Знание о расстоянии между числами поможет избежать классических грубых ошибок. Например, если вы делаете итеративное вычисление, например, поиск корней уравнения, то нет смысла задавать точность выше, чем это расстояние. Если вы зададите точность меньше этого, то цикл итерации будет длиться бесконечно.

Поскольку числа с плавающей точкой – лишь приближение к действительным числам, то при их использовании практически всегда будет присутствовать неточность. Эта неточность, называемая ошибкой округления, может приводить к удивительным результатам. Когда вы вычитаете близкие по значению числа, то наиболее значимые цифры взаимовычтутся, а наименее значимые (те, где ошибка скрывается ошибка округления) передвинутся на более значимые позиции, «загрязняя» дальнейшие вычисления (этот эффект получил название «размазывание», «smearing»). Вам нужно следить за применяемыми алгоритмами, чтобы не допустить этого. Например, представьте решение уравнения $x^2 - 100000x + 1 = 0$. Поскольку один из корней

требует такого вычитания $-b + sqrt(b^2 - 4)$, то можно вычислить сначала другой $r_1 = -b + sqrt(b^2 - 4)$, а потом найти первый по формуле $r_2 = 1/r_1$, потому что для любого ax2 + bx + c = 0 корни удовлетворяют условию $r_1 r_2 = c/a$.

Размазывание может случиться и в гораздо более тонких моментах. Представьте библиотеку, вычисляющую e^x по формуле $1 + x + x^2/2 + x^3/3! + ...$. Это работает для положительных x, однако для больших отрицательных работать перестанет. Четные степени дадут большие положительные числа, и вычитание отрицательных вообще не окажет влияния на результат. Проблема здесь в том, что ошибка округления для больших положительных чисел оказывается значительно больше, чем правильный ответ. В результате вычисление по этой формуле даст в ответе бесконечность! Решение же очень простое – для отрицательных x вычислять $e^x = 1/e^{|x|}$.

И конечно же, даже и говорить не стоит, что числа с плавающей точкой нельзя использовать для финансовых вычислений. Для них специально разработаны десятичные классы (decimal) в языках вроде Python и C#. Числа с плавающей точкой предназначены для эффективных научных вычислений. Однако эффективность бесполезна без нужной точности, поэтому помните о возможных ошибках округления!

Автор оригинала - Chuck Allison

Читайте код

(В оригинале - Read Code)

Мы, программисты, странные существа. Нам нравится писать код. При этом нам не нравится его читать. Конечно же, писать код гораздо интереснее, а читать код порой тяжело, а иногда практически невозможно. Читать код других людей – нелегкая задача. И необязательно из-за того, что их код плохо написан, а просто из-за того, что они по-другому мыслят и решают задачу другими способами, чем вы. А задумывались ли вы когда-нибудь, что чтение чужого кода может повысить ваш собственный уровень?

В следующий раз, когда вам придется читать чей-то код, задумайтесь на мгновение. Легко ли читать этот код, и если нет, то почему? Плохое форматирование? Несоответствующие имена? Несколько сущностей перемешаны друг с другом? Возможно, ограничения языка делают невозможной хорошую читаемость? Старайтесь учиться на чужих ошибках, чтобы не делать таких же самому. Вы можете обнаружить весьма неожиданные вещи. Например, техника уменьшения зависимостей может в качестве побочного эффекта сделать код более сложночитаемым. А то, что одни люди назовут элегантным кодом, другие назовут нечитаемым.

Если код легко читаем, посмотрите внимательнее, есть ли в нем что-нибудь полезное для вас. Возможно, это паттерн проектирования, о котором вы не слышали ранее или же не знали как его реализовать. Или же функции написаны более компактно и их имена более информативны, чем ваши. Некоторые open source проекты полны примеров того, как надо писать качественный, легко читаемый код, при этом другие являются примерами полной противоположности. Загрузите себе код нескольких из них и взгляните внутрь.

Чтение своего собственного старого кода может быть также весьма поучительным. Возьмите какой-нибудь свой старый проект и загляните внутрь. Возможно, вы обнаружите, что написать этот код было гораздо проще, чем теперь его прочесть. Ваш ранний код может даже вызвать волну стыда, вроде того, когда вы узнаете от кого-то о том, что именно вы говорили, напившись в баре. Посмотрите на это с другой стороны – оцените то, насколько вы профессионально выросли за эти годы, это может быть весьма мотивирующим. Найдите сложные для понимания фрагменты старого кода и задумайтесь, возможно, вы все еще делаете те же самые ошибки и сейчас.

Так что в следующий раз, когда вы захотите повысить свой уровень мастерства в программировании, не читайте еще одну книгу. Читайте код.

Автор оригинала - Karianne Berg

Чтобы улучшить код, удалите его

(В оригинале - Improve Code by Removing It)

Лучше меньше, да лучше. И как бы избито это не звучало, иногда это действительно так.

Одно из улучшений, которое я сделал с нашим кодом в течении последних недель, было удаление значительных его кусков.

Мы писали софт, следуя принципам XP, в том числе и YAGNI (You Aren't Going to Need It — «Вам это не понадобится»). Однако люди есть люди, и в нескольких местах мы потерпели неудачу.

Я заметил, что отдельные задачи выполняются слишком долго. Простые задачи, которые должны были бы выполняться практически мгновенно. Это происходило из-за того, что они были избыточно реализованы, с кучей свистулек и колокольчиков, потребности в которых не было, однако они были добавлены, потому что тогда добавить их казалось хорошей идеей.

Так что я упростил код, повысил производительность и снизил уровень энтропии всего кода, удалив проблемные куски. К счастью, юнит-тесты подтвердили, что при удалении ничего другого не сломалось.

Простой и безусловно полезный опыт.

Итак, почему же ненужный код оказывается в репозитории? Почему программист чувствует необходимость писать лишний код, и почему этот код проходит через фазу ревью? Практически всегда причины следующие:

- Это было прикольно, и программист хотел это сделать. *(Подсказка: Пишите код, потому что он добавляет ценность, а не потому что он вас развлекает)*.
- Кто-то решил, что это понадобится в будущем и поэтому лучше всего закодировать это прямо сейчас. (Подсказка: Это не соответствует принципу YAGNI. Если вам эта функциональность не нужна прямо сейчас, то и не надо ее прямо сейчас реализовывать).
- Это не казалось чем-то большим и сложным, поэтому было проще это реализовать, чем идти к заказчику выяснять, надо ли им это или нет. *(Подсказка: Написать и потом сопровождать написанное в любом случае сложнее. И с заказчиком всегда можно договориться. Маленький кусочек кода имеет тенденцию превращаться в снежный ком, требующий серьезного сопровождения)*.

- Программист «придумал» дополнительные требования, которые не были ни задокументированы, ни обсуждены, что оправдало новую функциональность. Ведь требования – это все. *(Подсказка: Программисты не должны составлять требования, это задача заказчика)*

Да, над чем вы сейчас работаете? Вы уверены, что это точно необходимо?

Автор оригинала - Pete Goodliffe

Шаг назад - и автоматизируйте, автоматизируйте, автоматизируйте!

(В оригинале - Step Back and Automate, Automate, Automate)

Я работал с программистами, которые, когда их просили подсчитать количество строк кода в проекте, вручную открывали код в текстовом редакторе и суммировали значение «количество строк» для каждого файла. Через некоторое время они вновь повторяли эту процедуру, когда их снова об этом просили. И еще раз. И еще. И это было плохо.

В другой раз я работал на проекте, имеющем громоздкий процесс сборки, включающий помещение результата на сервер и множество кликов мышкой. Однажды этот процесс был автоматизирован, после чего скрипт запускался сотни раз, гораздо чаще, чем предполагалось изначально. И это было хорошо.

Итак, почему люди делают одну и ту же рутинную работу множество раз вместо того, чтобы вернуться на шаг назад и найти время на ее автоматизацию?

Первое частое заблуждение: автоматизация – это только для тестирования.

Безусловно, автоматизация тестирования – супер, но зачем на этом останавливаться? Повторяющиеся задачи есть в любом проекте: контроль версий, компиляция, сборка, генерация документации, отчеты. Для большинства этих задач скрипт будет работать лучше, чем мышка. Да и выполнение скучной работы делается быстрее и надежнее

Второе частое заблуждение: у меня есть IDE, мне не нужна автоматизация.

Вы когда-нибудь слышали аргумент «Но ведь это работает (компилируется/запускается/проходит тест) на моей машине» от своих коллег? Современные IDE имеют тысячи настроек, и невозможно обеспечить, чтобы у каждого в команде эти настройки были идентичными. Системы автоматизации сборки вроде Ant или Autotools обеспечивают вам повторяемость эксперимента.

Третье частое заблуждение: мне придется изучить экзотические инструменты, чтобы реализовать автоматизацию.

Вы можете многого добиться со встроенным в ОС языком (например, bash или PowerShell) и системами автоматизации сборки. Если необходимо взаимодействовать с web-сайтами, используйте инструмент вроде iMacros или Selenium.

Четвертое частое заблуждение: я не смогу автоматизировать это потому что я ничего не могу сделать с этим форматом файлов.

Если в вашем процессе присутствуют документы Word, таблицы или изображения, автоматизация действительно может оказаться непростой. Но – действительно ли вам нужны именно эти форматы? Можете ли вы использовать обычный текст, текстовые таблицы, XML, инструмент генерации рисунка из текстового файла? Иногда небольшая настройка процесса может дать серьезный выигрыш в снижении рутины.

Пятое частое заблуждение: у меня нет времени в этом всем разобраться.

Вам не нужно изучить весь bash или Ant, чтобы начать. Изучайте в процессе использования. Как только у вас появится задача, которую, вам кажется, можно автоматизировать, изучите только то, что нужно для ее автоматизации. И сделайте это как можно раньше, ведь в начале проекта найти «лишнее» время проще. И как только у вас получится, вы (и ваш босс) увидите, что в автоматизацию стоит инвестировать ресурсы.

Автор оригинала - Cay Horstmann

Юникс-утилиты - это ваши друзья

(В оригинале - The Unix Tools Are Your Friends)

Если бы мне пришлось выбирать между IDE и юниксовыми утилитами, я бы выбрал утилиты не задумываясь ни секунды. Ниже я постарался изложить причины, почему вам стоит ими овладеть в совершенстве.

Во-первых, IDE часто являются инструментом для одного конкретного языка, а юниксовые утилиты могут работать со всем, что представлено в текстовой форме. В современном мире, когда каждый год появляются новые языки, изучение работы в стиле Юникс – это инвестиция, способная многократно окупиться.

Далее, если IDE – это лишь набор команд, которые его разработчики сочли нужными, то при помощи юниксовых утилит вы можете сделать все, что захотите. Представляйте их как конструктор Лего и собирайте нужную вам команду из маленьких, но многогранных утилит. Например, вот так можно рассчитать сигнатуру Каннингема – «образ» файла, от которого остались только точки с запятой, скобки и кавычки, что может многое сказать о его содержимом.

```
for i in *.java; do
    echo -n "$i: "
    sed 's/[^"{};]//g' $i | tr -d '\n'
    echo
done
```

И еще, каждая операция в IDE относится к единственной задаче, например, добавить новый шаг в отладочную сборку. Для сравнения, оттачивание мастерства во владении утилитами Юникса повышает вашу эффективность для всех задач сразу. Например, я применил sed для преобразования сборки проекта для компиляции на многопроцессорной архитектуре.

Утилиты Юникса разрабатывались в эру, когда многопользовательский компьютер имел 128 килобайт памяти. Гениальность их дизайна означает в настоящее время возможность обрабатывать огромные массивы данных с невероятной эффективностью. Большинство утилит работают как фильтры, работая с обычной строкой, что означает отсутствие верхнего предела на количество данных, которые могут быть обработаны. Хотите узнать количество записанных изменений в полутерабайтной английской википедии? Не проблема:

```
grep '<revision>' | wc –l
```

без труда выдаст вам ответ. Если вы сочтете последовательность команд полезной, вы легко сможете запаковать ее в скрипт, используя уникальные возможности вроде перенаправления данных в циклы и условия. Более того, юникс-команды, исполняемые в конвейерах, как в примере выше, могут распределять нагрузку по нескольким процессорам в современных системах.

Изящность и открытость кода юниксовых утилит делают их общедоступными, даже на платформах с ограниченными ресурсами, как например, медиа плейер или DSL роутер. Вряд ли у подобных устройств будет полноценный графический интерфейс, но они часто содержат BusyBox, предоставляющий практически все часто используемые утилиты. Если вы пишете под Виндоуз, то Cygwin также предоставит вам все возможные утилиты как в исполняемом виде, так и в исходных кодах.

И наконец, если ни одна из утилит не удовлетворяет вас, то вы легко можете расширить их набор. Просто напишите программу на любом языке, придерживающуюся нескольких простых правил: программа должна читать данные в виде текстовых строк со стандартного входа и должна выдавать результаты «без прикрас» на стандартный выход. Параметры, влияющие на работу программы, должны передаваться через командную строку. Всего лишь соблюдайте эти правила.

Автор оригинала - Diomidis Spinellis

Языки предметной области

(В оригинале - Domain-Specific Languages)

Когда бы вы не вслушались в разговор экспертов в какой-либо предметной области, будь то шахматисты, работники детского сада или страховые агенты, вы всегда заметите, что они используют слова, отличающиеся от обычных, используемых повседневно. Это то, что называется «язык предметной области» (далее для краткости ЯПО) – набор слов и выражений, описывающий вещи, характерные для данной предметной области.

В мире программного обеспечения под ЯПО подразумевается набор исполняемых выражений на языке, специфичном для данной области, с ограниченным словарем и грамматикой, которые при этом читабельные, понятные и (хорошо если так) используемые экспертами предметной области. ЯПО предназначен для разработчиков или научных работников, находящихся в этой предметной области долгое время. Например, язык конфигурационных файлов Unix или языки, созданные при помощи LISP-макросов.

ЯПО принято разделять на *внутренние* и *внешние*.

- **Внутренние ЯПО** создаются при помощи языков программирования, чей синтаксис близок к обычному языку людей. Сделать такое проще на языках, предлагающих больше «синтаксического сахара», как например Ruby или Scala, и сложнее на тех, которые такого не предоставляют (например, Java). Большинство внутренних ЯПО представляют собой обертки для существующих API, библиотек, предоставляя менее головоломный вариант использования существующей функциональности. Их можно исполнять напрямую. В зависимости от реализации и домена, ЯПО могут создавать структуры данных, определять зависимости, запускать процессы и задачи, связываться с другими системами или проверять пользовательский ввод. Синтаксис внутренних ЯПО ограничивается языком программирования, на котором они реализованы. Существует много шаблонов, например, построители выражений, цепочечные методы, аннотации, которые могут помочь вам «привести» ваш язык программирования к вашему ЯПО. Если используемый язык программирования не требует перекомпиляции, то внутренний ЯПО может быть весьма быстро разработан при непосредственном участии эксперта предметной области.

- **Внешние ЯПО** – это текстовые или графические выражения языка, причем текстовые, как правило, используются чаще. Текстовые выражения могут обрабатываться инструментарием, включающим лексический анализатор, грамматический разбор, модификатор модели, генераторы и другие виды постобработки. Внешние ЯПО часто преобразуются во внутреннюю модель, представляющую базис для дальнейшей обработки. Хорошо помогает задать грамматику (например, расширенную форму Бэкуса-Наура). Грамматика предоставляет начальную точку для всего остального (редактор, визуализатор, грамматический разборщик). Для простых ЯПО ручной разборщик грамматики может быть достаточным (например, можно использовать регулярные выражения). В более сложных случаях лучше посмотреть в сторону специально разработанных для работы с грамматиками инструментов, таких как openArchitectureWare, ANTlr, SableCC, AndroMDA. Определение внешнего ЯПО как диалекта XML тоже часто встречается, хотя читабельность такого языка часто является проблемой, особенно для тех, кто не знаком с XML.

Вы всегда должны отслеживать, для кого предназначен ваш ЯПО. Для разработчиков, менеджеров, заказчиков или конечных пользователей. Вам нужно адаптировать технический уровень языка, доступные инструменты, помощь по синтаксису, раннюю проверку, визуализацию для уровня тех, кто будет этим пользоваться. Скрывая технические детали, ЯПО может помочь пользователям, давая им возможность настраивать систему под себя без помощи разработчиков. Это может также ускорить разработку путем распределения работы после того, как ЯПО займет свое место в системе. ЯПО может постепенно развиваться. Существуют различные способы миграции уже существующих выражений и грамматик.

Автор оригинала - Michael Hunger