

УДК 82-82+82-83+82-84+004.42

ББК 87.52+87.53+32.973

Л80

Лот, Алексей Сергеевич

Полезные конспекты книг и авторские заметки по информационным технологиям. Без формул [Приложение к петербургскому литературному журналу «МОСТ»] / Сост. А.С. Лот. – СПб.: Век искусства, 2025. – 145 с.

В этой книге уважаемый читатель найдёт множество советов по конструированию программного кода, общим вопросам, возникающим при работе в agile-команде, поисковой оптимизации веб-сайтов (SEO), автороведческой экспертизе и безопасности паролей. Книга составлена из полезных тезисов, выписанных автором из двенадцати печатных технических книг на русском языке.

ISSN 1991-7023

УДК 82-82+82-83+82-84+004.42

ББК 87.52+87.53+32.973

Книга содержит упоминание организаций, запрещённых на территории РФ: «Фейсбук».

© 2024, 2025, составление, А.С. Лот

© 2025, оформление, А.С. Лот

Для отзывов и пожеланий автору: windowsisloading@yandex.ru

«Век искусства»: ladolad@inbox.ru

Подписано в печать 03.02.2026.

Формат 145 x 200. Бумага офсетная. Усл. печ. л. 2.5. Тираж 150 экземпляров.

Алексей Лот

**Полезные конспекты книг
и авторские заметки
по информационным технологиям.
Без формул**

Москва – Санкт-Петербург
2025

***Публикация книги посвящается
научным руководителям автора
Александру Николаевичу Алфимцеву,
Юрию Николаевичу Павлову,
Олегу Гарегиновичу Петросяну.***

Полезные высказывания из книги «Agile для всех» Лемея

В разделе приведены цитаты из [1].

Agile – термин, описывающий схожие по сути, но различные по методикам подходы. Самый популярный Agile-подход – это скрам:

1) работа разбивается на двухнедельные периоды, называемые спринтами;

2) в конце каждого периода должно быть что-то действительно завершенное и готовое для выпуска;

3) в течение каждого спринта проводится ежедневная планерка (daily standup) или скрам-митинг (scrum meeting);

4) во время митинга каждый член команды делится тем, что он завершил, над чем работает и что может препятствовать прогрессу;

5) идея получать что-то готовое каждые две недели – стимул для повышения производительности и укрепления морального духа;

6) необходимость личного общения каждое утро – способ улучшить коммуникацию в команде (выявить разногласия);

7) приоритизация и передача результатов в двухнедельные циклы позволяют объективировать противоречия в видении продукта высокого уровня;

8) утренние планерки показывают отставание по срокам;

9) цель – повысить эффективность команды;

10) скрам дает набор рабочих процедур для структурирования деятельности компании.

Agile требует соответствия предпринимаемых действий достигаемым целям, ставя вопрос о том, почему сотрудники, команды и организации делают так, а не иначе. Agile предполагает выполнение большего объема работ за меньшее время при условии привлечения больших ресурсов: энергии, открытости, готовности честно размышлять над трудными вопросами. Agile призывает превращать предположения в факты.

Agile можно применять во всех отраслях. Нет лучшего или правильного подхода к Agile.

Принципы и ценности «Почему» – Практики «Как» – Результаты реального мира «Что» – Принципы и ценности «Почему».

Если мы чувствуем, что не соответствуем «Почему», то можем менять «Как», а если «Как» не срабатывает для «Что», то нужно пересмотреть «Почему». Сначала нужно понять, почему возникла потребность в изменении привычного способа работы. Основополагающие принципы Agile:

1) мы начинаем с наших клиентов;

2) мы сотрудничаем на ранних стадиях и часто;

3) мы планируем неопределенность.

Манифест Agile:

- люди и взаимодействие важнее процессов и инструментов;
- работающий продукт важнее исчерпывающей документации;
- сотрудничество с заказчиком важнее согласования условий контракта;

тракта;

- готовность к изменениям важнее следования первоначальному плану.

Важное здесь не отрицает остального. Mindset – образ мышления.

2001 – основание Agile-движения.

1995 – Scrum.

90-е – Crystal.

1999 – XP.

2005 – LeSS.

2011 – SAFe.

Agile Alliance: Agile – способность создавать изменения и реагировать на них, чтобы добиться успеха в неопределенной и турбулентной среде. Ошибочный путь наименьшего сопротивления:

- люди в организации будут избегать работы с клиентами, если она выходит за рамки их повседневных обязанностей и не приносит бонусов;

- люди в организации будут отдавать предпочтение работе, которую могут выполнить с наименьшими усилиями, не покидая собственную команду или отдел;

- работа над текущим проектом будет продолжаться до тех пор, пока не вмешается самый старший из тех, кто его одобрил.

В Agile команда выдает готовый результат за каждый временной блок (time box). В конце каждого временного блока собирается обратная связь от предполагаемой аудитории, и на ее основе планируется следующий набор временных блоков, называемых итерацией. В реальном мире идеальный Agile недостижим. Движения наподобие Agile отвечают на один и тот же вопрос: «Как организации могут адаптироваться к потребностям клиентов в быстро меняющемся мире?» Движения отличаются метриками успеха организации:

Agile – скорость (velocity), с которой команда выпускает продукты;

Lean – количество потерь, которое команда исключает из производства;

Design Thinking – количество ценности, которую продукты команды предоставляют клиентам.

При использовании трех подходов важность метрик определяется ранжированием движений в организации. Чтобы не менять подходы Agile

один за одним, нужно до выбора подхода задаваться вопросом: «Какие у нас цели и как наш способ работы помешал их достигнуть?», а после: «Какие Agile-ценности и принципы помогут достичь конкретных целей?». Клиентоориентированность означает думать о потребностях, целях и опыте, прежде чем думать о конкретной вещи, которую мы собираемся доставить клиентам. Потребности и цели коллег, руководителей и клиентов могут не совпадать, но их нужно примирять. Гибкость измеряется способностью меняться и развиваться в зависимости от потребностей клиента, а не скоростью выполнения работ. Dogfooding – разработка версии приложения для тестирования и внутреннего использования. Вопросы в Agile нужно ставить с точки зрения клиента.

Сотрудники организации должны работать с клиентами напрямую. Клиенты выбирают тот опыт, который наилучшим образом соответствует их потребностям и целям, а не тот, который является самым «инновационным» или «прорывным». Во втором принципе Agile сотрудничество на ранних стадиях означает сотрудничество во время как стратегических, так и тактических переговоров, открывая возможность поиска новых и неочевидных решений, что требует открытости, восприимчивости и готовности делиться идеями; частое сотрудничество означает продолжение обсуждений на протяжении всего процесса создания и реализации, обеспечивая согласованность стратегии и тактики, чтобы получить больше возможностей для корректировки курса по мере необходимости. Модель Spotify – многоуровневая многофункциональная система, где люди работают на уровне «отрядов», «кланов», «отделов», «гильдий». Нужно поддерживать неформальное общение друг с другом, а не ждать совещания. На каждое совещание должно отводиться строго определенное количество времени – идея таймбоксинга. При встраивании в рабочий процесс сначала задать вопрос: «На каком мы этапе? Кто занимался решением этой задачи?» Ежедневные планерки должны длиться не более пятнадцати минут, во время которых все стоят на ногах и отвечают на вопросы:

- что я сделал вчера, чтобы помочь команде разработки достичь цели спринта?

- что я сделаю сегодня, чтобы помочь команде разработки достичь цели спринта?

- какие препятствия мешают мне или команде разработки достичь цели спринта?

Ретроспектива – это совещание в конце спринта или подведение итогов по проекту, на котором команда обменивается мнениями по поводу проведенной совместной работы и определяет изменения, которые можно внести при подготовке к следующему спринту или проекту, но не

критикует проведенную работу. Ретроспектива дает командам возможность выстроить общее понимание цели и ответственности в рамках работы.

Основные вопросы: «Как вы считаете, наш рабочий процесс помогает нам воплощать в жизнь наши методы и достигать наших целей?», «Каким образом мы будем действовать в следующий раз?»

Post mortem – собрания, во время которых участники открыто обсуждают ошибки, не боясь наказания. Ретроспектива дает возможность менять Agile-методы команды. Сворачивание проекта означает, что вы открыты к возможным изменениям клиентов и не собираетесь тратить дополнительные ресурсы на проект, которому не суждено стать успешным.

Абсолютные количественные параметры оценки Agile-методов превратят Agile в бессмысленный список задач.

Большинство людей предпочитают определенность страданий страданиям неопределенности.

Названия принципов Agile: клиентоориентированность, сотрудничество, открытость к переменам. Необязательно использовать все три принципа.

Нельзя скрывать от руководителей плохие новости.

Процесс крайне прост: голова, сердце, руки.

Нужно отслеживать скорость рынка чаще скорости внутри компании.

Полезные высказывания из книги «Е-mail маркетинг» Демина

В разделе приведены цитаты из [2].

Сегодня, чтобы человек совершил покупку, потенциальный клиент должен увидеть вашу рекламу, новость о вас или ваш логотип 24 раза.

Главная цель e-mail маркетинга – не просто продажа товаров (услуг), а формирование доверия и лояльности клиента.

Сформируйте доверие; покажите, что вы – профессионал в своем деле, поделитесь интересной, ценной или полезной информацией, и продажи неуклонно пойдут вверх.

E-mail маркетинг приносит денег больше, чем любая другая реклама.

Чем подробнее сегментирована база по различным характеристикам клиентов, тем выше продажи.

Для рассылок нужно использовать специальные почтовые сервисы, чтобы обычный почтовый ящик не заблокировали.

Бесплатный сервис до 2000 подписчиков – MailChimp.

Единственное, где понадобится бюджет, – сбор активных подписчиков.

В рассылках можно писать новости, обзоры удачных и неудачных сайтов, выдавать свой опыт по частям, делиться фотоотчетами.

Выбирают часто тех, кто в рассылке показал свою деятельность и уровень навыков.

Рассылка позволяет не зависеть от рекламных агентств, промоутеров и продавцов.

Схема работы e-mail маркетинга:

1. Стратегия и задачи, оценка своего нынешнего состояния (о клиенте, о проблеме).
2. Форма регистрации/подписки.
3. Одностраничник.
4. Привлечение подписчиков.
5. Выбор сервиса для отправки почты.
6. «Пряник» за регистрацию.
7. Виды писем: тестовые письма, письма-подтверждения, транзакционные письма, автописьма, цепочки писем, триггеры, разовые письма.
8. «Мягкие» продажи.
9. Усиление накала.
10. Акция или специальное предложение.
11. Сегментация.

12. Замедление, выдача контента.

Чтобы люди вас читали и ждали писем, рассылка должна решать конкретную проблему или ряд проблем и быть написана для одного человека или нескольких.

Составьте портрет каждого вашего потенциального клиента, максимально близкие к конкретному человеку.

Критерии сегментации:

- имя;
- пол;
- семейное положение;
- если женат/замужем – кто супруг (а);
- дети;
- какие отношения с детьми;
- чем человек занимается, на что живет;
- в чем его самая большая неудовлетворенность;
- какое самое сильное внешнее желание (человек говорит об этом; для бизнесмена – расширение бизнеса);
- сокровенные желания (чего хочет на самом деле); например, лежать и ничего не делать;
- кто раздражает и сердит.

Затем ответьте, какую проблему этого человека решает ваша рассылка.

В интернете взаимодействуют с контентом не более 1% аудитории (ставят лайки, оставляют комментарии).

Анализируйте работу конкурентов и учитесь у них.

Почтовые сервисы предоставляют код формы подписки для размещения на своем сайте.

Максимально эффективно можно использовать имя клиента, e-mail и номер телефона.

В среднем лишь 30% оставленных номеров реальные.

Всплывающие окна с формами подписки повышают конверсию в среднем на 30%.

На «приземляющей» странице обычно есть текст о товаре (услуге), фотографии товара или клиентов, отзывы, видеоролики, графики, форма подписки или заказа. Нет лишь ссылок на другие страницы и ресурсы, поэтому одностраничники дают самую высокую конверсию.

Большие одностраничники делают с целью продажи чего-либо.

Заголовок «приземляющей» страницы должен быть конкретным и содержать конечную выгоду – это первое, на что обращают внимание посетители.

До 80% эффективности любой рекламы зависит именно от заголовка.

На одностраничном сайте ничто не должно отвлекать клиента от совершения задуманного вами действия: ни ссылки на другие сайты, ни странные картинки, ни заумный текст. Если вы сомневаетесь в необходимости какого-то элемента, смело его удаляйте.

Незамысловатые, простые и близкие человеку видео вызывают намного больше доверия, чем рекламные ролики.

Можно рассказать о товаре своими словами, продемонстрировать главные преимущества.

Описывайте не сам продукт, а преимущества, которые благодаря ему получит ваш клиент.

Посетитель должен понимать, что ему нужно сделать, чтобы воспользоваться вашим предложением: ввести свои данные, нажать кнопку, позвонить. Отсутствие такого элемента может снизить эффективность страницы до 70%.

Отзывы реальных клиентов повышают доверие к вам и вашей организации: они могут быть в виде текста или видео. Для максимального эффекта от текста нужна фотография человека, оставившего отзыв, его имя и название организации, в которой он работает. Идеально, если будет некликабельный адрес сайта. Каждый отзыв начинается с подзаголовка – это может быть ударная фраза или главные преимущества товара (услуги).

Маркированные списки – это краткое описание товара. Короткие фразы привлекают внимание, быстро читаются и дают понять, чем вы отличаетесь от конкурентов.

Кнопки соцсетей помогут пользователю поделиться ссылкой на ваш одностраничник и так привлечь новых клиентов.

Дизайном пренебрегать не стоит. Графики, диаграммы и инфографика отлично привлекают внимание и удерживают читателя.

Нужно составить техзадание:

- дизайнер делает сам дизайн, подбирает картинки, цветовую гамму, шрифты и так далее;

- копирайтер наполняет страницу продающими текстами, которые буквально заставляют купить товар (услугу);

- программист собирает все пазлы в единую картину (можно обойтись услугами веб-дизайнера).

Есть сервисы создания одностраничников.

Самые популярные сервисы по статистике сайта:

- Google analytics;

- LiveInternet;

– «Яндекс. Метрика».

Если на сайт зашли 1000 человек, а подписались на рассылку 100 – значит, конверсия страницы – 10%.

Способы привлечения подписчиков:

1. Статьи в СМИ (в них максимум указать название организации, себя как автора и ссылку на сайт).

2. «Вирусный» маркетинг.

3. Форумы – создать тему с бесплатной консультацией на свою тему.

4. «Твиттер».

5. «Фейсбук» (Запрещён на территории РФ).

6. «Ютуб» – запись максимум 2–3 минуты.

7. Паблик в «ВКонтакте».

Чем уже тематика, тем более качественных подписчиков получите.

8. Купонные сайты.

9. Отзывы.

10. Subscribe.

11. Сервисы ответов.

12. Обмен ссылками, SEO-оптимизация. wordstat.yandex.ru

13. Википедия.

14. Собственная партнерская программа.

15. Ссылка с призывом в подписи.

16. PodFM.

17. Социальные комментарии к вашему сайту или страничке (в соц-сети).

18. Клиенты на «живых» мероприятиях.

19. Официальные документы, книги, договора, счета и тому подобное.

20. Доменное имя.

21. Всплывающие окна PopUp.

22. RSS-лента.

23. Создайте материал, которым хочется поделиться.

24. Реклама подписки в других тематических рассылках.

25. Ссылка на форму подписки в счете.

26. Офлайн-подписка.

27. Предложение о подписке на визитке.

28. QR-код.

29. Конкурсы, викторины, розыгрыши.

30. Ограничьте ваше предложение.

31. Просто попросите распространить.

32. Сосредоточьтесь на интересах существующих подписчиков.

33. Учитывайте конфиденциальность данных подписчиков.

«Яндекс. Директ», Google AdWords.

Реклама: в почтовой выдаче, на тематических площадках. Именно контекстная реклама – один из самых простых способов сразу получить горячих подписчиков.

CTR.

Ищите слова, которые еще никто не додумался использовать для объявлений.

Подписка через VK только для массового потребителя.

Большая часть информации в соцсетях воспринимается визуально.

Картинка в рекламном посте и на приземляющей странице должны иметь что-то общее.

Первым откликом на рекламу в соцсети могут быть гневные отзывы.

Пряник – бесплатность за подписку.

Кнут – любой прием, который вынуждает человека действовать незамедлительно.

Можно начать продажу в первом письме. Профессиональные сервисы рассылок: emailVision, exactTarget, intelligentEmails.

Полупрофессиональные сервисы рассылок: Pechkin-mail, SmartResponder, mailigen, mailChimp, epochta, Subscribe, justclick, unisender.

Письма с подтверждениями люди открывают чаще всего. Не забывайте о «взрослении» аудитории.

Принципы (элементы) копирайтинга:

1. Заголовок должен заставить человека прочесть первый абзац. Шрифты в заголовке отлично привлекают внимание и вызывают больше доверия. Ровным числам – 10, 20, 30 – люди доверяют меньше, чем неровным – 15, 17, 21.

2. Подзаголовок раскрывает суть заголовка.

3. Короткое первое предложение.

4. Заголовки параграфов помогают быстро оценить текст.

5. Первый абзац – простой и понятный.

6. Снимок или рисунок (текст с изображением больше привлекает внимание).

7. Подпись под изображением – мощная фраза.

8. Описание продукта.

9. Технические характеристики.

10. Отзывы, свидетельства, награды (нельзя подделывать!).

11. Цена.

12. Вид обратной связи для вопросов.

13. Предугаданные возражения.

14. Итоги.
15. Призыв к действию (чего хотите от читателя?).
16. Причина реагировать сейчас.
17. Способы оплаты – можно платежный агрегатор.
18. Гарантия.
19. Постскриптум.

Максимальное внимание привлекает первый квадрат 500 на 500 пикселей.

Не более 3-х шрифтов в письме.

Самые читаемые – четырех-пятистрочные абзацы.

Оптимальная ширина письма – 600 пикселей, или 60– 80 символов 10–12 pt.

Фон в письме – белый или светлых оттенков.

С HTML можно создать уникальный дизайн или фирменный стиль писем, чтобы сразу понимали, что речь идет о вашей компании, можно вставить в письмо кнопки соцсетей.

С картинкой в письме обязательно должен быть текст.

Вопросы для проверки своего письма: что хотите сообщить заказчику; почему вам должно быть это интересно; что вы хотите, чтобы сделал подписчик; куда попадают ваши подписчики после переходов по ссылкам в письме; не перегружает ли дизайн текст или другие объекты.

Проверять письмо в разных браузерах до отправки и проверять ссылки.

Рассылать разную рекламу для проверки реакции на содержимое.

B2B – на вы.

B2C – на ты или вы.

Обязательна ссылка в письме, но не более 3-х.

Рассылка должна быть регулярной.

Чем больше о вас знают потенциальные клиенты, тем лучше.

В 2016 году в мире будет отправляться более 192 млрд писем в день.

Разбивать аудиторию хотя бы на 3 группы.

Чем больше вы знаете о подписчиках, тем лучше.

Переподписка – раз в несколько месяцев.

В e-mail маркетинге важны взаимоотношения с клиентом, а не хитрости и уловки.

Отписываться будут всегда.

Чем больше неактивных клиентов, тем чаще письма будут попадать в спам, поэтому неактивных лучше удалять самостоятельно.

Рассылку надо планировать на 3–6 месяцев.

Анализировать отношение открытий к переходам.

Полезные высказывания из книги «Чистый код» Мартина

В разделе приведены цитаты из [3].

Единственный способ выдержать график – постоянно поддерживать чистоту в коде.

Поэтому нужно развивать «чувство кода».

Чистый код:

- элегантный и эффективный;
- логика прямолинейна;
- зависимости – минимальные;
- обработка ошибок – полная;
- производительность – близкая к оптимальной;
- решает одну задачу;
- поддерживается в чистоте с течением времени.

Следующая книга – «Agile software development; Principles, patterns, and practices», 2002.

Содержательные имена:

– имена всех объектов должны отвечать на все главные вопросы: почему существует, что делает и как используется, не требуют дополнительных комментариев, содержательные, передают намерения программиста;

- код очевиден, контекст следует из самого кода;
- не содержат ложных ассоциаций, затемняющих смысл кода;
- не используются слова со скрытыми значениями, отличными от предполагаемого;

– лучше обойтись без кодирования типа контейнера в имени.

Имена не содержат малозаметных различий, непохожи друг на друга излишне.

Имена не дезинформируют.

0 не равен O, 1 не равна I.

Используется правильный шрифт.

Используются осмысленные различия.

Имена не дублируют зарезервированные слова с незначительными изменениями.

Если имена различаются, они должны обозначать разные понятия. Различия имен содержательны.

Префиксы используются, если создают осмысленные различия.

Не содержат неинформативных, избыточных слов.

Имена не отличаются только суффиксами.

Читателю кода понятен смысл различающихся имен – нет соблазна использовать 2 похожих имени по одному назначению.

Имена удобопроизносимы.

Программирование – социальная деятельность.

Не должны состоять из одних сокращений.

Состоят преимущественно из слов разговорной речи.

Удобны для поиска.

Легко находимы в большом объеме кода.

Относительно редкие.

Длинные имена лучше коротких.

Однобуквенные используются только для локальных переменных в коротких методах.

Длина имени соответствует размеру его области видимости.

Имя не содержит информации о типе или области видимости.

Не создает хлопот при расшифровке.

Неразумно заставлять каждого нового работника изучать очередной «язык» кодирования.

Имя остается понятным даже в случае опечатки.

Имя не усложняет изменение типа переменной.

Типы в именах не кодируются.

Переменные классов без префиксов.

Классы и функции компактны – можно обойтись без префиксов.

Имя не содержит балласта.

Имена интерфейсов без префиксов.

Имена не содержат лишней информации.

Не заставляют мысленно преобразовывать имена в другие.

Используются имена из пространств задачи и решения.

Счетчик цикла с малой областью видимости можно назвать 1 буквой – это традиция.

Ясность превыше всего.

Код понятен для других людей.

Имена классов и объектов – существительные и их комбинации без глаголов, не содержат обобщенных понятий – Manager, Processor, Data, Info.

Имена методов – глаголы или глагольные словосочетания.

Методы чтения или записи и предикаты образуются из значения и префикса get, set и is.

При перегрузке конструкторов использовать статические методы-фабрики с именами, описывающими аргументы (принудительное использование таких методов).

Нет остроумных шуток.

Ясность предпочтительнее развлекательной ценности.

Нет просторечий и сленга.

Нет шуток.

Одно слово для каждой концепции.

Имена функций законченные и логичные.

Нет эквивалентных методов с именами `fetch`, `retrieve`, `get` в разных классах.

Нет `controller`, `manager`, `driver` в одной кодовой базе.

Имена различны принципиально.

Единый согласованный лексикон.

Не используется одно слово в двух смыслах.

Две разные идеи не обозначены одним термином.

Нет каламбура.

Мысли в коде выражаются доступно.

Используются имена из пространства решения: термины из области информатики, названия алгоритмов, паттернов, математические термины, технические имена.

Используются имена из пространства задачи (клиентские): если нет подходящего программиста, узнаются у специалиста в предметной области.

Разделение концепций из пространств задачи и решения.

Несодержательные сами по себе имена помещаются в определенный контекст для читателя кода – классы, функции и пространства имен с правильно выбранными названиями.

В крайнем случае контекст уточняется префиксом.

Контекст не должен вычисляться.

Функции разделяются на меньшие смысловые фрагменты.

Нет избыточности контекста.

Нет работы против собственного инструментария.

Короткие имена лучше длинных, если только их смысл понятен читателю кода.

Имена экземпляров более точные.

Развивать описательные навыки и единый культурный фон.

Не должно быть опасения возражений при переименовании.

Функции:

Первый уровень структуризации.

Длина не избыточна.

Не содержит повторяющихся фрагментов кода.

Один уровень абстракции.

Функции компакты.

Функции еще компактнее.

Функции желательно не более 20 строк.

Все функции предельно очевидны.

Блоки в командах if, else, while и так далее должны состоять из 1 строки, в которой обычно – вызов функции.

Функции не содержат вложенных структур.

Не более 1–2 отступов.

Функция должна выполнять только одну операцию. Она должна выполнять ее хорошо. И ничего другого она делать не должна.

Если функция выполняет только те действия, которые находятся на одом уровне под объявленным именем функции, то эта функция выполняет одну операцию.

Функции пишутся прежде всего для разложения более крупной концепции (иначе говоря, имени функции) на последовательность действий в следующем уровне абстракции.

Чтобы определить, что функция выполняет более 1 операции, надо попробовать извлечь из нее другую функцию, которая бы не являлась простой переформулировкой реализации.

Функцию, выполняющую только одну операцию, невозможно осмысленно разделить на секции.

Все команды функции находятся на одном уровне абстракции.

За каждой функцией должны следовать функции следующего уровня абстракции.

Switch, длинные цепочки if-else скрывать в низкоуровневом классе и не дублировать в коде (использовать полиморфизм).

Принцип единой ответственности (single responsibility principle).

Принцип открытости-закрытости (open-closed principle).

Программа не содержит неограниченного количества других функций с аналогичной структурой (можно использовать абстрактную фабрику).

Имя точно описывает, что делает функция.

Длинное имя функции лучше короткого невразумительного.

Не бойтесь расходовать время на выбор имени функции.

В именах функций использовать те же словосочетания, глаголы и существительные, что и в модулях.

В идеальном случае количество аргументов функции равно нулю.

Использовать функции 1 аргумента:

- для проверки некоторого условия, связанного с аргументом;
- для обработки аргумента, его преобразования и возвращения;
- для события (вход есть, выхода нет);
- должно быть предельно ясно, что перед читателем событие;
- остальных форм функций с 1 аргументом лучше избегать;
- не использовать аргументы-флаги.

Бинарные функции оправданны, если оба аргумента – упорядоченные компоненты одного значения.

Использовать все доступные способы для сведения функций к унарной форме.

Аргументы должны иметь естественную связь и естественный порядок.

Хорошо подумать перед созданием тернарной функции.

Упаковывать аргументы в объекты.

Если переменное количество равноправных аргументов – упаковать в List.

Хорошее имя функции способно объяснить смысл функции, порядок и смысл ее аргументов.

В унарных функциях функция и аргумент должны образовывать естественную пару «глагол – существительное».

Функция не делает чего-то скрытно от пользователя.

Нет побочных временных привязок функции.

Нет побочных эффектов.

Нет причин лишней раз обращаться к сигнатуре функции (нет повторных заходов).

Выходных аргументов следует избегать.

Функция может изменять состояние только владельца.

Функция либо что-то делает (команда), либо отвечает на какой-либо вопрос (запрос).

Функция либо изменяет состояние объекта, либо возвращает информацию об этом объекте.

Исключена неоднозначность имен функций.

Функции-команды не возвращают коды ошибок.

Вместо возвращения кодов ошибок используются исключения.

Тела блоков try/catch выделены в отдельные функции.

Функции выполняют 1 операцию.

Обработка ошибок – одна операция.

Нет магнитов зависимостей – классов или перечислений, импортируемых и используемых многими другими классами.

Нет дублирования алгоритмов.

Уменьшена вероятность ошибки.

Goto не используется.

Много return, break, continue допускается в компантных функциях.

В коде сначала излагаются мысли, а затем «причесываются».

Система рассматривается как история, которую нужно рассказать.

Комментарии – неизбежное зло.

Только код может правдиво сообщить, что он делает.

Свести использование комментариев к минимуму.
Комментирование – причина повысить качество кода.
Вместо юридических комментариев – ссылки на них.
Информацию лучше передавать в имени функции.
Использовать комментарии для информации о намерении.
Комментарии – в случае неудобочитаемых форм данных.
Комментарии для предупреждения о последствиях.
Комментарии TODO на будущее.
Комментарии для подчеркивания важности обстоятельства.
Не делать комментарии на скорую руку.
Комментарий не приводит к поиску расшифровки в других модулях.
Использовать аналог Javadoc.
Не комментировать бормотанием.
Не комментировать избыточно.
Комментарии точнее кода.
Комментарии точные и соответствуют коду.
Комментарий не вводит в заблуждение и не дезинформирует.
Бессмысленные или обязательные комментарии исключены.
Комментарий не повышает риск обмана и недоразумений.
Длинные журналы комментариев исключены.
Комментарии не загромождают и не усложняют код.
Комментарии-шумы исключены.
Комментарии не утверждают очевидное, не предоставляя новой информации.
Комментарии не бесполезны.
Комментарии не вызывают желания игнорировать их.
Комментарии не содержат эмоций.
Комментарии делают работу приятной и эффективной.
Комментарии не используются там, где можно использовать функцию или переменную.
Заголовки в комментариях применяются, когда приносят ощутимую пользу.
Закрывающие скобки не комментируются.
Ссылки на авторов в комментариях заменяются использованием системы контроля версий.
Нет закомментированного кода.
Нет HTML-комментариев.
Комментарии описывают код, к которому отнесены.
Комментарии не содержат дискуссий, исторических подробностей, не относящихся к делу.
Связь между комментарием и его кодом очевидна.

Цель комментария – объяснить код, который не объясняет сам себя.
Комментарий не нуждается в объяснении.

Комментарии для API общего пользования не помещаются в коде, не предназначенном для общего потребления.

Комментарий упрощает понимание алгоритма пользователем.

Код должен быть хорошо отформатирован.

Форматирование облегчает передачу информации.

Маленькие файлы понятнее больших.

Типичная длина файла 200 строк, предел – 500.

Исходный файл выглядит как статья.

Имя файла простое, но содержательное.

Имени файла достаточно, чтобы определить, этот модуль нужен или нет.

Начальные блоки файла описывают высокоуровневые концепции и алгоритмы.

Степень детализации увеличивается к концу файла.

В конце файла собираются все функции и подробности низшего уровня.

Код читается слева направо и сверху вниз.

Законченные мысли отделяются пустыми строками.

Строки кода с тесной связью должны быть сжаты по вертикали.

Тесно связанные концепции не находятся в разных файлах.

Следует избегать запущенных переменных.

Читателю не нужно прыгать туда-сюда по исходным файлам и классам.

Переменные объявляются максимально близко к месту использования.

Переменные перечисляются в начале каждой функции.

Управляющие переменные циклов объявляются внутри конструкции цикла.

Переменные экземпляров объявляются в начале класса.

Если одна функция вызывает другую, то они располагаются вблизи друг друга по вертикали.

Вызывающая функция располагается над вызываемой.

Концептуально родственные фрагменты кода располагаются близко друг к другу по вертикали.

Важные концепции излагаются сначала с минимальным количеством второстепенных деталей.

Строки делать по возможности короткими.

Пробелы для визуального обозначения приоритета операторов.

Длинные списки – причина для разделения на классы.

Горизонтальное выравнивание не применяется.

Отступы выделяют области видимости.

Не применяются вырожденные области видимости.

Группа разработчиков согласует единый стиль форматирования.

Код продукта оформлен в едином стиле.

Внешний пользователь не в курсе деталей реализации.

Методы интерфейса устанавливают политику доступа к данным.

Классами предоставлены абстрактные интерфейсы, посредством которых пользователь оперирует с сущностью данных.

Пользователь не имеет представления о фактическом формате данных.

Объекты скрывают свои данные за абстракциями и предоставляют функции, работающие с этими данными.

Структуры данных раскрывают свои данные и не имеют осмысленных функций.

Процедурный код (код, использующий структуры данных) позволяет легко добавлять новые функции без изменения существующих структур данных.

ООП-код упрощает добавление новых классов без изменения существующих функций.

Процедурный код усложняет добавление новых структур данных, так как требуется изменение всех функций.

ООП-код усложняет добавление новых функций, так как требуется изменение всех классов.

Данные необязательно представляются в виде объектов.

Закон Деметры – модуль не должен знать внутреннее устройство объектов, с которыми работает.

Метод *f* класса *C* должен ограничиваться вызовом методов следующих объектов: *C*; объекты, созданные *f*; объекты, переданные *f* в качестве аргумента; объекты, хранящиеся в переменной экземпляра *C*.

Метод не должен вызывать методы объектов, возвращаемых любыми из разрешенных функций.

Не использовать цепочки вызовов (разделять).

Не используются гибриды объектов и структур данных.

Разные уровни детализации не смешиваются.

Объекты передачи данных – *Data transfer object* – классы с открытыми переменными без функций используются для преобразования низкоуровневых данных, получаемых из базы, в объекты кода приложения.

Bean – компоненты из приватных переменных, операции с которыми осуществляются при помощи методов чтения и записи, преимуществ не имеют.

Активные записи – active records – структуры данных с открытыми переменными, а также с навигационными методами – это структуры данных и бизнес-логику не содержат.

Обработка ошибок не заслоняет логику программы.

Ошибки обрабатываются стильно и элегантно.

Используются исключения вместо кодов ошибок.

Начинать с написания команды try-catch-finally для кода, способного вызывать исключение.

Тип исключения сужается до реально иницируемого.

Блоки try должны напоминать транзакции.

Использовать непроверяемые исключения.

С исключениями передавать содержательные сообщения об ошибках.

Из сведений об исключении должно быть понятно, с какой целью выполнялась операция.

Классы исключений определены в контексте потребностей вызывающей стороны.

Инкапсулированы вызовы сторонних API.

Для обработки особых случаев использовать паттерн особый случай.

Вместо null выдается исключение или особый случай.

Для API, возвращающего null, – делать обертки.

Не возвращать null из методов.

Не передавать null при вызове методов.

Запрещать передачу null по умолчанию.

Чистый код должен быть надежным.

Вместо приведения типа контейнера лучше использовать параметризованные контейнеры.

Ограничить передачу граничных интерфейсов по платформе (можно инкапсулировать).

Для стороннего кода писать тесты.

Сторонний код тестировать в рамках понимания его работы.

Конструкторы по умолчанию должны иметь конфигурацию.

Писать учебные тесты, граничные тесты.

Можно заранее определять интерфейсы, затем писать паттерн-адаптер к готовым.

Для граничного кода необходимо четкое разделение сторон и тестов, определяющие ожидания пользователя.

Количество обращений к границам стороннего кода сводится к минимуму.

Законы TDD:

– не пишите код продукта, пока не напишете отказной модульный тест;

– не пишите модульный тест в объеме большем, чем необходимо для отказа (невозможность компиляции является отказом);

– не пишите код продукта в объеме большем, чем необходимо для прохождения текущего отказного теста.

Тесты не уступают в качестве коду продукта.

Тесты развиваются вместе с продуктом.

Модульные тесты обеспечивают гибкость, удобство сопровождения и возможность повторного использования кода.

Без тестов любое изменение становится потенциальной ошибкой.

Некачественные тесты приводят к некачественному коду продукта.

Чистый тест характеризуется удобочитаемостью: ясностью, простотой и выразительностью.

В тестах использовать паттерн «построение – операции – проверка».

Тесты не делают ничего лишнего, в них используются только действительно необходимые типы и функции.

Использовать наборы функций и служебных программ, использующих API.

Код тестов не такой эффективный, как код продукта.

Чтобы избежать дублирования, можно воспользоваться паттерном шаблонный метод.

Не более 1 assert в функции теста.

Одна концепция в тесте (1 тест – 1 проверка).

Характеристики чистых тестов:

– тесты должны выполняться быстро;

– тесты не зависят друг от друга;

– тесты дают повторяемые результаты в любой среде.

Результатом выполнения теста должен быть логический признак (результат очевиден).

Тесты создаются своевременно непосредственно перед кодом продукта.

Класс должен начинаться со списка переменных. Сначала перечисляются открытые статические константы. Далее следуют приватные статические переменные.

За ними идут приватные переменные экземпляров.

Затем открытые функции.

Приватные вспомогательные функции, вызываемые открытыми функциями, непосредственно за самой открытой функцией (газетная статья).

Открытые переменные обычно не используют.

Предпочтительно объявлять переменные и вспомогательные функции приватными.

Иногда переменную или вспомогательную функцию приходится объявлять защищенной, чтобы иметь возможность обратиться к ней из класса.

Ослабление инкапсуляции должно быть последней мерой.

Классы должны быть максимально компактными.

Компактность класса определяется его ответственностью.

По имени класса можно определить его размер.

Краткое описание класса должно укладываться в 25 слов, без выражений «если», «и», «или», «но».

Принцип единой ответственности: SRP – класс или модуль должен иметь одну и только одну причину для изменения (одну ответственность).

Система должна состоять из множества мелких классов со сформированной структурой.

Класс взаимодействует с другими классами для реализации желаемого поведения системы.

Классы должны иметь небольшое количество переменных экземпляров.

Чем с большим числом переменных работает метод, тем выше связность этого метода со своим классом. Создавать классы с высокой связностью не рекомендуется.

Высокая связность означает, что методы и переменные класса взаимозависимы.

Рост числа переменных экземпляров свидетельствует о необходимости выделения класса.

Рефакторинг может удлинить программу.

Приватные методы, действие которых распространяется на небольшое подмножество класса, – признак возможности усовершенствований.

Структурирование проводится с учетом изменений.

Время, необходимое для понимания класса, падает почти до нуля.

Вероятность того, что одна из функций нарушит работу другой, ничтожно мала.

Принцип открытости-закрытости: классы должны быть открыты для расширений, но закрыты для модификации.

Структура системы должна быть такой, чтобы обновление системы создавало как можно меньше проблем.

В идеале новая функциональность должна реализовываться расширением системы, а не внесением изменений в существующий код.

С помощью интерфейсов и абстрактных классов класс изолируется от конкретных подробностей.

Принцип обращения зависимостей: классы системы должны зависеть от абстракций, а не от конкретных подробностей.

В программных системах фаза инициализации, в которой конструируются объекты приложения и «склеиваются» основные зависимости, должна отделяться от логики времени выполнения, получающей управление после ее завершения.

Инициализация – область ответственности.

Код инициализации пишется системно и отделен от логики времени выполнения.

Удобные идиомы не ведут к нарушению модульности.

Приложение ничего не знает о main или о процессе конструирования.

Все аспекты конструирования помещаются в main или в модули, вызываемые из main.

Если момент создания объекта должен определяться приложением, то использовать паттерн «Фабрика».

Вся информация о конструировании хранится на стороне main.

Приложение изолировано от подробностей построения.

Использовать внедрение зависимостей Dependency Injection

или обращение контроля Inversion of Control для отделения конструирования от использования.

Итеративная разработка – сегодня реализуем текущие потребности, а завтра перерабатываем и расширяем систему для реализации новых потребностей.

Архитектура программных систем может развиваться последовательно, если обеспечить правильное разделение ответственности.

Компонент-сущность (entity bean) – представление реляционных данных в памяти.

Применять АОП – модульные конструкции, называемые аспектами, определяют, в каких точках системы поведение должно меняться некоторым последовательным образом в соответствии с потребностями определенной области ответственности (изменения вносит инфраструктура без необходимости вмешательства в целевой код).

Посредники (proxies) хорошо подходят для простых ситуаций – вызова методов отдельных объектов или классов.

Использовать POJO-объекты.

DAO – Data accessor object – объект доступа к данным.

Использовать aspectJ.

Не полагаться на BDUF.

На каждом уровне абстракции архитектура должна оставаться простой и обладать минимальными привязками.

Хороший API должен исчезать из вида большую часть времени.

Один человек не может принять все необходимые решения.

Принятие решений лучше всего откладывать до последнего момента.

Стандарты применяются разумно, когда они приносят очевидную пользу.

Главная задача – реализовать интересы клиента.

Использовать DSL (их код читается как структурированная форма текста, написанного экспертом в данной предметной области).

Используйте самое простое решение из всех возможных.

Четыре правила простой архитектуры:

- архитектура обеспечивает прохождение всех тестов;
- не содержит дублирующегося кода;
- выражает намерения программиста;
- использует минимальное количество классов и методов.

Система должна делать то, что задумано ее проектировщиком.

Существует простой способ убедиться в том, что система действительно решает свои задачи.

Система, тщательно протестированная и прошедшая все тесты, контролируема.

Обеспечение полной контролируемости системы повышает качество проектирования.

Для системы необходимо написать тесты и постоянно выполнять их.

Рефакторинг – последовательная переработка кода.

Рефакторинг проводится при наличии полного набора тестов.

В системе не дублируется реализация.

Применять повторное использование даже в мелочах.

Дублирование – главный враг системы.

Код системы возможно понять без глубокого понимания решаемой проблемы.

Постараться сделать код выразительным.

Неравнодушие – драгоценный ресурс.

Использовать прагматичный подход взамен бессмысленного догматизма.

Применять нагрузочное тестирование.

Многопоточность – стратегия устранения привязок.

Многопоточность – аналогия работы нескольких компьютеров.

Многопоточность повышает быстродействие не всегда.

Многопоточность может изменить архитектуру.

При многопоточности нужно предусмотреть проблемы многопоточной взаимной блокировки, одновременное обновление.

Многопоточность требует больше производительности и кода.

Правильная реализация многопоточности сложна.

Ошибки в многопоточности обычно не воспроизводятся.

Многопоточность обычно требует фундаментальных изменений в стратегии проектирования.

Предусмотреть перебивание потоками друг друга (требуется знание обработки байт-кода и атомарных операций модели памяти).

Многопоточные архитектуры должны отделяться от основного кода.

Код реализации многопоточности имеет собственный цикл разработки, модификации и настройки.

При написании кода многопоточности возникают специфические сложности.

Предусмотреть обработку обращений к общему объекту.

Инкапсулировать данные: жестко ограничить доступ и область видимости общих данных.

Вместо использования общего объекта каждому потоку можно предоставить копию.

Использовать синхронизацию.

Потоки должны быть как можно более независимы.

Постараться разбить данные на независимые подмножества, с которыми могут работать независимые потоки (возможно, на разных процессорах).

Использовать потоково-безопасные коллекции.

Использовать неблокирующие решения.

Изучать доступные классы на предмет потоково-безопасности.

Модели логического разбиения поведения программы при многопоточности:

– производители-потребители: потоки-производители создают задания и помещают в буфер или очередь. Потоки-потребители извлекают задания из очереди и выполняют их. Производители перед записью ждут появления свободного места в очереди, а потребители ждут появления заданий в очереди. Производитель записывает задание и сигнализирует о том, что очередь не пуста. Потребитель читает задание и сигнализирует о том, что очередь не заполнена. Обе стороны готовы ждать оповещения о возможности продолжения работы;

– модель читатели-писатели: писатели пишут в общий ресурс, который считывают читатели. Писатель может блокировать читателей. Нужно найти баланс между потребностями читателей и писателей, что-

бы обеспечить правильный режим работы, нормальную производительность и избежать зависания;

– модель обедающих философов: за круглым столом сидят философы-потоки и думают, в центре – тарелка еды. Каждому философу для еды доступно 2 вилки-ресурсы – по 1 от соседей. Когда философ проголодается – берет вилки, поел – кладет обратно. Сложности проектирования – взаимные блокировки, обратные блокировки, падение производительности и эффективность работы.

Изучать базовые алгоритмы, разбираться в решениях.

Избегать использования нескольких методов одного совместно используемого объекта.

Избегать зависимостей между синхронизированными методами.

Или использовать 3 стандартных решения: – блокировка на стороне клиента;

– блокировка на стороне сервера;

– адаптирующий сервер.

Код не должен перегружаться лишними синхронизированными объектами, так как блокировки создают задержки и увеличивают затраты ресурсов.

Синхронизированные секции должны иметь минимальные размеры.

Корректное завершение не может быть бесконечным ожиданием потока.

Реализовать логическую изоляцию конфигураций многопоточного кода.

Протестировать программу с количеством потоков, превышающим число процессоров.

Применять инструментовку кода для повышения вероятности сбоев.

Сначала отлаживать основной код.

Не игнорировать системные ошибки, считая их случайными разовыми сбоями.

Убедиться, что сам код работает вне многопоточного контекста, созданием РОЮ-объектов, вызываемых из потоков.

Реализовать многопоточный код так, чтобы он мог выполняться в различных конфигурациях: с разным числом потоков, тестовых заместителей, времени работы тестов, количеством повторов тестов.

Найти средства измерения производительности системы в разных конфигурациях.

Реализовать систему так, чтобы количество программных потоков могло легко изменяться, в том числе во время работы системы, в том числе с автоматическим регулированием количества потоков.

JVM не гарантирует вытесняющей многопоточности.

Протестировать систему во всех средах, которые могут использоваться для ее развертывания, начиная с ранней стадии.

Заставить поток исполняться по разным путям в тесте методами `object.wait()`, `object.sleep()`, `object.yield()`, `object.priority()` – инструментовка кода.

Автоматическая инструментовка с использованием `AspectOriented Framework`, `CGLIB`, `ASM`, `ConTest`.

Суть тестирования – сломать предсказуемость пути выполнения.

Использовать стратегию случайного выбора пути выполнения для выявления ошибок.

Типичные источники многопоточных ошибок: работа с общими данными из нескольких потоков, использование пула общих ресурсов.

Использовать длительное тестирование многопоточного кода перед включением в систему.

Программирование ближе к ремеслу, чем к науке.

Спасать положение нужно именно сейчас.

Во время переработки кода не добавлять в программу новые возможности.

Предусмотреть, в каких местах будет появляться новый код.

Множество разных типов со сходными методами – причина выделить класс.

TDD: не вносить в систему изменения, нарушающие работоспособность системы.

Добавление теста или класса ничего не нарушит.

Удалять бесполезные функции.

Тесты всегда должны хотя бы запускаться.

Прочитать последовательное очищение.

Переработка кода напоминает кубик Рубика.

Важный аспект хорошей архитектуры – логическое разбиение кода.

Плохой код тянет группу ко дну.

Открытый код требует смелости и доброй воли.

Полезные высказывания из книги «Отладка приложений для Microsoft .Net» Джона Роббинса

В разделе приведены цитаты из [4].

Большинство команд тратит в среднем 50% цикла разработки на отладку.

Отладка требует специального обучения.

Для того чтобы эффективно отлаживать любую технологию, нужно знать намного больше, чем может дать книга, сфокусированная на конкретной технологии.

Книги по программированию бывают посвящены менеджменту и технологиям.

Основные программы отладки. NET: VisualStudio и WinDBG.

Разработчики должны использовать только учетные записи, обладающие привилегиями пользователя.

john@wintellect.com – автор.

Ошибки помогают понять работу вещей.

Ошибки в ПО могут привести к смене работы.

Ошибка – все что угодно, что заставляет пользователя страдать.

Категории ошибок: аварии и зависания, низкая производительность и масштабируемость, неверные результаты, нарушения безопасности, противоречивые пользовательские интерфейсы, неудовлетворенные ожидания.

Нельзя выпускать на рынок продукт с авариями и зависаниями.

Windows Error Reporting.

Пользователи иногда перестают пользоваться продуктом из-за одного неудачного опыта.

С точки зрения управления проектами главное – уделить внимание производительности.

Должны быть заданы конкретные цифры, связанные с производительностью.

Не делать продукт более медленным, чем его предыдущая версия.

Тестировать приложения по сценариям, наиболее точно отражающим реальный мир.

Наборы данных из реального мира брать у клиентов.

Реальные данные должны быть модифицированы – удалена конфиденциальная информация.

Писать код проверки результатов.

Выставлять требования к производительности, масштабируемости, безопасности.

Проводить тестирование безопасности и моделирование угроз.
Интерфейс приложения не должен противоречить интерфейсу среды.

Приложение не противоречит сочетаниям клавиш среды запуска.
«Designing web usability: the practice of simplicity» Якоб Нильсен.
«Don't make me think! A common sense approach to web usability»

Стивен Круг.

cnn.com – лучший пример дизайна.

joelonsoftware.com/articles.

Все члены команды должны посещать клиентов и наблюдать за использованием ПО.

Никогда не обещать того, чего не сможете дать, и всегда реализовывать обещанное.

Категории причин появления ошибок: слишком короткие или нереальные сроки выполнения; подход «сначала код, потом подумаем»; неправильное понимание требований; невежество разработчиков или недостаточное качество обучения; наплевательское отношение к работе.

Учитывать время на обучение, необходимое для того, чтобы реализовать какую-либо функцию.

Команда разработчиков должна быть истинным хозяином своего расписания, определять реалистичные даты выпуска за счет сокращения числа функций.

Перед написанием кода хорошенько подумать об архитектуре.

Продумать все «что, если».

Определить все риски проекта.

Члены команды не должны отдавать контроль над конструированием системы не умеющим это делать людям.

Не начинать сразу кодировать при получении плана.

Должна быть реалистичная оценка технологии и план разработки еще до включения компьютера.

Исключить добавление новых функций в планировании, которые изначально не планировались.

Чем больше деталей будет обнаружено в ПО и продумано до начала кодирования, тем меньше будет ошибок.

Нарисовать пользовательский интерфейс и полностью проработать каждый сценарий использования.

Коридорное тестирование.

Инженеры должны проявлять желание изучать предметную область.

Лучшие инженеры – это не те, кто умеет жонглировать битами, а те, кто качественно решает проблему пользователя.

Разработчики должны знать достаточно хорошо операционную систему, язык, технологию, которые используются в проектах.

Эффективно использовать выделяемые на обучение средства, проанализировав сильные и слабые стороны команды.

Лучший способ узнать что-то о технологии – сделать что-либо с применением этой технологии.

Все, о чем в действительности заботится ваш менеджер, – это возможность ежедневно сообщать своему боссу, чем вы занимаетесь день за днем.

Настоящие инженеры проникнуты глубокой гордостью за то, что они производят, и хотят тратить время и усилия на всех этапах разработки.

Компании и люди с реальной приверженностью качеству демонстрируют множество общих черт: тщательное предварительное планирование, личную ответственность, жесткий контроль качества и отличные коммуникационные навыки.

Только те, кто уделяет внимание деталям, выпускают продукты вовремя и с отличным качеством.

Проводить ревизии эффективности работы ежемесячно.

Регистрировать число ошибок в продукте ежемесячно (общее число обнаруженных за месяц).

«Software reliability: measurement, prediction, application» Джон Мьюз.

В среднем коде содержится одна ошибка на каждые 10 строк.

«Code complete» МакКоннелл.

По мере того как продукт создается, цена исправления ошибки растет экспоненциально, как и цена отладки.

Ускорять отладку и тестирование на этапе планирования.

Хороший отладчик = хороший разработчик.

Самая важная черта отладчика – интуиция.

Чтобы превратиться в отличного отладчика, необходимо быть специалистом в следующих областях: ваш проект, ваш язык, ваша технология/инструменты, ваша операционная система/среда.

Должна быть хорошая документация или объяснение на 15 мин от разработчиков, чтобы лучше узнать проект.

Нужно знать, что используемый язык делает за сценой.

Необходимо четко представлять себе, где можно найти более подробную информацию на случай, если она понадобится.

Писать утилиты.

Показывать свой код интервьюерам – сразу попадаем в первые 20%.

Изучение кода других инженеров и добавление новых функций – хорошая практика.

Читать код framework.

Друзья и коллеги-инженеры – лучшие источники знаний о разработке и отладке.

Подход к отладке:

Воспроизведите ошибку.

Опишите ошибку.

Всегда предполагайте, что это ваша ошибка. Разделяйте и властвуйте.

Думайте творчески.

Используйте инструменты.

Начните тяжелую отладку.

Убедитесь, что ошибка исправлена. Извлеките урок и поделитесь им с другими.

Полезные высказывания из книги «Надежный код» Бруно

В разделе приведены цитаты из [5].

Более высокий уровень абстрагирования позволяет больше времени уделять важным элементам любого проекта.

Нет единого мнения по вопросу, что именно должны знать и уметь все разработчики.

Разработка ПО – это не инженерия.

Настройка производительности и безопасности не должна откладываться на конец проекта.

После прочтения советов необходимо выработать привычку их применения.

На практике методология водопада работает не очень хорошо, поскольку многие важные особенности программы проявляются лишь на этапе реализации.

Нереалистично тестировать ПО в конце цикла разработки.

agilealliance.org.

agilemanifesto.org/principles.html.

Сейчас используются Scrum (наиболее широко применяется в Microsoft), XP, TDD.

blogs.msdn.com/e7.

Шаги разработки:

- сбор сведений от заказчика, совместное обсуждение требований, детальное тестирование, описание параметров компонентов и общей архитектуры системы;

- процедуры контроля качества кода – стандарты, совместная разработка, оптимизация, модульное тестирование;

- обширное тестирование системы и сборки.

Методы контроля качества применять как можно раньше.

Чем больше кода написано, тем сложнее его тестировать.

Стремиться и сосредоточиться на качестве, надежности, безопасности.

Применять итеративную разработку с итерациями не более 6 –8 недель, полностью завершать один компонент перед началом работы, разделять работы на несколько автономных групп, достаточно часто обмениваться информацией о состоянии проекта.

Использовать типичные методологии развертывания, оборудования и инструментов, планировать процессы, стремясь к предсказуемости и повторяемости процедур.

Для ускорения процессов создания и развертывания, упрощения обмена информацией использовать в командах общие инструменты и

процедуры для управления исходным кодом, сборки, ведения баз данных, общие подходы к программированию и единую терминологию.

Отказаться от модели водопада.

Создавать проверяемые модули.

Ежедневно запускать сборку и запускать автоматические BVT-тесты.

Использовать продукты компании в ее работе, начиная с самых ранних стадий их разработки.

Идентифицировать участников приложения (типы) и способы их взаимодействия друг с другом (прототипирование; концепции, взаимоотношения и взаимодействия проверяются на полноту и корректность; здесь же анализируются риски дизайн).

В приложениях на управляемом коде в качестве языка метапрограммирования, позволяющего изменять поведение приложения во время выполнения, применять XML.

Производительность с самого начала часть любого проекта.

Факторы масштабирования – часть дизайна приложения.

Безопасность закладывается в дизайне приложения.

Тестеры гарантируют полное покрытие кода тестами.

Учиться эффективно управлять памятью.

Использовать безопасное программирование.

Сердцем программирования являются этапы анализа требований и проектирования.

Цикл разработки ПО: – анализ требований;

– проектирование;

– спецификации;

– программирование;

– тестирование;

– развертывание;

– обслуживание.

Написанию кода обязательно должен предшествовать этап проектирования.

Дизайн приложения в идеале не зависит от особенностей реализации.

В ООП 40–50% времени разработчик тратит на проектирование кода.

Число написанных строк кода – неверный показатель труда.

Секрет успеха заключается в неизменности цели.

На устранение проблемы на этапе тестирования требуется в 4 раза больше времени, чем на этапе проектирования.

C# – прямой потомок C++.

Существительные из предметной области могут стать объектами, глаголы и глагольные группы – поведением.

Проектирование должно управлять реализацией, но не наоборот.

Из списка объектов предметной области удаляются не взаимодействующие с другими.

Использовать UML для моделирования в проектировании.

Типы UML-схем см. в книге.

Изоляция классов друг от друга – один из принципов ООП.

В конструкторе классов VS изменения в схеме классов немедленно отражаются в коде и наоборот.

Один из способов повысить гибкость кода – писать его как можно меньше.

Метаданные хранить в xml-файлах конфигурации в корневом каталоге приложения (не использовать ini и реестр).

Использование метаданных позволяет разработчику отделить конкретные логические детали от исходного кода.

Настройки, применимые ко всем приложениям на данном компьютере, хранятся в machine.config.

Храните в метаданных настройки приложения.

Храните в метаданных данные приложения.

Управляйте поведением приложения при помощи метаданных.

Все компоненты должны при возможности управляться конфигурацией.

Изменение файлов конфигурации не требует регрессионного тестирования.

Использовать сжатие данных, передаваемых по сети.

Не использовать множество маленьких статических изображений.

Стандарт HTTP/1.1 предполагает одновременную загрузку браузером только двух ресурсов с одного имени хоста (новые браузеры игнорируют это требование).

Не злоупотреблять перенаправлением страниц.

В веб-приложении не стоит использовать много уникальных имен хостов, чтобы избежать излишних запросов DNS.

Максимально сокращать объем данных, передаваемых по сети, для оптимизации производительности.

Добивайтесь максимально эффективного использования пропускной способности сети.

Свести число HTTP-запросов к минимуму.

Используйте сжатие HTTP.

Минимизируйте объем кодов JavaScript и CSS (Yui Compressor, JsMin).

Уменьшайте палитру изображений.

Настраивайте сроки действия данных (Expires, Cache-Control, ETag).

Прочитать High Performance WebSites (Souders), mnot.net/cache_docs.

Увеличение числа параллельных TCP-портов позволит параллельно загружать больше содержимого.

HTTP/1.0 требовал создания нового соединения для каждого HTTP-запроса. Пакеты keep-alive позволяют браузерам отправлять несколько HTTP-запросов с одного соединения, что сокращает сетевой трафик за счет уменьшения числа открываемых и закрываемых соединений (заголовков Connection: keepalive).

Запросы к DNS могут занимать до 120 мс.

Уменьшение числа обращений к DNS (рекомендуется ≤ 4 – 6 уникальных имен хостов) улучшит время ответа страницы, но может негативно повлиять на параллельную загрузку содержимого.

Если у приложения мало ресурсов, в общем случае лучше обойтись одним именем хоста.

В общем случае перенаправлений лучше избегать, используя лишь при необходимости поддерживать в течение некоторого времени старый URL или создать URL с более запоминаемым адресом страницы.

Используйте сети доставки содержимого (CDN).

Внедряйте спрайты CSS – вместо загрузки одного изображения загружайте несколько параллельно (для мелких значков используйте один спрайт).

В общем случае внедрение JavaScript и CSS в страницу ускоряет прорисовку.

Размещение JavaScript и CSS во внешних файлах позволяет кешировать их браузером (при последующих обращениях внешние сценарии выгоднее).

Размещайте CSS в начале страницы (если файлы CSS расположены вне блока HEAD, браузер заблокирует прогрессивную прорисовку).

Размещайте сценарии JavaScript в конце страницы (при загрузке файлов JS браузеры блокируют загрузку любого другого содержимого, включая содержимое, идущее по параллельным TCP-портам).

Программа совершенствования производительности: – определить сценарии использования и приоритеты;

- проанализировать разработки конкурентов;
- сформулировать задачи производительности;
- внедрять эффективные методики;
- измерять и тестировать.

Устанавливайте ограничения производительности для ключевых сценариев (например, количество запросов get).

Непрерывно анализируйте и тестируйте производительность.

Экспериментируйте и разбирайтесь в поведении пользователя (измененная версия приложения предоставляется небольшой группе пользователей).

Упреждающе загружать страницы и сценарии, о которых известно, что пользователь будет их посещать.

Выясните, какие факторы влияют на производительность работы конечного пользователя.

Применяйте инструменты тестирования производительности.

Приложение считается масштабируемым (scalable), если его производительность возрастает по мере расширения аппаратной базы.

Необходима возможность наращивать мощность приложения за счет инфраструктуры – новых серверов, накопителей данных и сетевого оборудования (работа пользователей при этом не должна зависеть от объема нагрузки).

Приложение должно быть всегда доступно для пользователей, поэтому нужно избавлять приложение от мелких сбоев наподобие выхода из строя жестких дисков, для чего нужно встраивать избыточность и отказоустойчивость как в дизайн веб-приложения, так и в топологию развертывания.

Чем больше оборудования и чем шире развернуто ПО, тем сложнее им управлять и обслуживать его, поэтому очень важно, чтобы разработчики приложения проектировали его таким образом, чтобы его просто было настраивать, развертывать и контролировать.

Производительность приложения необходимо проверять при различной нагрузке.

Веб-приложение можно на любом языке спроектировать так, что оно будет масштабируемым.

Разработчики должны учитывать возможность пиковых нагрузок и даже выбросов.

В дизайн приложения необходимо встроить возможность распределения нагрузки по нескольким серверам.

Масштабируемые приложения позволяют реагировать на повышение нагрузки простым добавлением нового оборудования.

Две основные стратегии масштабирования приложений при расширении аппаратной базы: вертикальное (scaling up) и горизонтальное (scaling out).

Вертикальное масштабирование состоит в добавлении ресурсов на тот же компьютер либо в переносе приложения на более мощный ком-

пьютер (как правило, модернизация компьютера заключается в установке дополнительных процессоров, добавлении памяти или увеличении дискового пространства). Преимущество вертикального масштабирования – простота. Со временем затраты на аппаратное обеспечение могут сделать использование приложения невыгодным. Может быть такое, что не удастся найти устройство требуемой мощности. Вертикальное масштабирование удобно в тех случаях, когда целевая аудитория приложения не слишком обширна и затраты на модернизацию не превышают выгоду от использования приложения.

В горизонтальном масштабировании по мере роста запросов приложения в систему включаются новые серверы (оборудование уникальной мощности не требуется). Преимущество горизонтального масштабирования – экономичность в сравнении с вертикальным. Кластер из большого количества дешевых серверов невысокой мощности может оказаться дешевле за счет низкой стоимости оборудования и пониженного энергопотребления (однако тут востребована отказоустойчивость, так как дешевое оборудование чаще выходит из строя). Горизонтальное масштабирование посредством более дорогих и качественных компьютеров обеспечит более устойчивую и надежную работу. В крупных системах предпочтение стоит отдавать горизонтальному масштабированию. Систему с горизонтальным масштабированием проще обслуживать. Основной недостаток горизонтального масштабирования – высокие затраты на администрирование и значительные накладные расходы, связанные с управлением десятками, сотнями и тысячами серверов (проблема решается API-интерфейсами администрирования). Горизонтальное масштабирование следует рассматривать в первую очередь для приложений со значительной целевой аудиторией и тенденцией к устойчивому росту.

В некоторых случаях горизонтального масштабирования разработчики сталкиваются со специфическими требованиями к дизайну, которые не поддаются линейному масштабированию.

При масштабировании уровня БД веб-приложения к вашим услугам вертикальное и горизонтальное масштабирования.

В MSSQL масштабирование выполняется созданием федерации и секционированием.

Выбирайте простую архитектуру, чтобы будущее расширение приложения не привело к недопустимым затратам на его обслуживание.

Выбирайте дизайн, пригодный для горизонтального масштабирования.

Используйте устройства для балансировки нагрузки (серверы можно добавлять и удалять в процессе работы).

Правильный выбор аппаратного обеспечения практически всегда связан с использованием однотипного оборудования на всех серверах определенного типа. Стандартизация оборудования не только экономит средства за счет оптовых закупок, но также снижает затраты на диагностику и обслуживание.

При сбое в ключевом компоненте приложение должно продолжать работу в сокращенном объеме (приложение должно быть отказоустойчивым).

Отказоустойчивость следует применять только к более важным компонентам.

На разработчике лежит ответственность за своевременное информирование руководства о важности создания отказоустойчивой инфраструктуры и о возможных вариантах ее создания.

Приложение должно проектироваться таким образом, чтобы минимизировать влияние зависимостей.

Сужение функциональности лучше, чем сообщение об ошибке.

Приложения должны быть просты в эксплуатации (приложение должно обладать управляемостью и сопровождаемостью).

Данные инструментирования может собирать отдельная служба.

Ведите активный мониторинг приложения.

Выделите ключевые метрики и цели мониторинга.

Планируйте стратегию роста и стратегию реакции на сбой.

Масштабирование «по ролям»: отдельные функциональные области масштабируются независимо друг от друга.

В каждом компоненте приложения предусматривайте аварийный выход.

Автоматизируйте основные задачи управления.

Составляйте план восстановления системы.

Постоянно оценивайте загруженность инфраструктуры и планируйте ее развитие.

«Writing secure code» Ховард.

«Threat modeling» Свидерски.

Используйте механизмы проверки подлинности и авторизации. NET (Windows Identity, Windows Principle, Generic Identity, Generic Principle).

Шифруйте конфиденциальные данные.

Считайте внешние приложения небезопасными по умолчанию.

Реализуйте безопасную обработку отказов.

Реализуйте безопасную обработку ошибок и исключений.

Приложения следует проектировать так, чтобы они могли выполняться с наименьшими возможными полномочиями (AntiCross Site Scripting Library).

Включайте в конфигурацию по умолчанию только необходимые компоненты.

По умолчанию приложение должно работать с ограниченными полномочиями.

Организируйте процесс сопровождения приложения и устранения ошибок (устранение сбоев и сбор отзывов).

Предоставьте пользователям руководство по установке и настройке.

Соблюдайте требования законодательства.

Организируйте диалог с пользователями по вопросам безопасности.

Разработайте план реагирования на проблемы с безопасностью.

«.NET Security Programming» Маршал.

Элементы политики безопасности времени выполнения:

В NET поддерживаются пользовательские разрешения и наборы разрешений.

Свидетельство источника предоставляется хостом, запрашивающим выполнение сборки.

Кодовая группа связывает свидетельство с набором разрешений.

Уровни политики определяют границы предоставляемых разрешений.

NET Framework Configuration Tool.

Code Access Security Policy Tool. Permissions View Tool.

Build Verification Testing.

Final Security Review.

Intrusion Detection Systems. FxCop.

В динамическом коде память выделяется динамическим объектам по требованию при помощи оператора new.

Модель управления памятью в NET опирается на два основных допущения: большие объекты живут долго и друг с другом чаще всего взаимодействуют объекты близкого размера.

Управляемая куча делится на поколения и кучу больших объектов. Поколения 0, 1 и 2 используются для группировки объектов по размеру и времени жизни. 2 – самое старое поколение с самыми большими объектами. 0, 1 – эфемерные поколения. Большими считаются объекты более 85000 байт. Большие объекты хранятся в куче больших объектов. Задача сборщика мусора – освободить память. Сборку мусора запускают события:

1. После очередного выделения памяти в случае его успешного завершения превышен порог для поколения 0. Новые объекты всегда помещаются в поколение 0. В поколении 0 всегда находятся самые молодые объекты.

2. В куче больших объектов не хватает памяти.

3. Вызван метод GC.Collect.

Сборщик мусора собирает сначала 0 поколение, затем 1 и 2.

Если обрабатывается какое-либо поколение, обязательно обрабатывается и младшее. Для объектов, оставшихся после сборки, сборщик повышает поколение. При сборке мусора объекты в обрабатываемых поколениях признаются недоступными. Дерево памяти при сборке мусора перестраивается. Объекты, не попавшие в дерево, считаются недостижимыми и готовыми к сборке. Сборщик мусора сжимает доступные объекты в управляемой куче. Использовать метод GC.Collect не рекомендуется, так как он способен помешать работе сборщика мусора в штатном режиме. Управляемый класс – это интерфейс между управляемым приложением и неуправляемыми ресурсами. Сборщик мусора будет искать в памяти только управляемый интерфейсный класс, не учитывая память для неуправляемого ресурса (GC.AddMemoryPressure и GC.RemoveMemoryPressure). Вызовите метод GC.AddMemoryPressure в конструкторе управляемого класса, чтобы применить нагрузку, равную памяти неуправляемого ресурса. В методе Finalize или Dispose вызовите GC.RemoveMemoryPressure.

Сборщик обрабатывает кучу больших объектов (LOH) при полной сборке мусора. Память в куче больших объектов не сжимается. Для крупных объектов лучше создавать буфер – это позволяет выделять для больших объектов смежные блоки памяти (вы экономите память, минимизируете фрагментацию и сокращаете количество полных (наиболее ресурсоемких) сборок мусора). При помощи Finalize () не стоит очищать управляемые объекты. Уничтожение объектов с дескриптором требует 2 цикла сборки мусора. В отличие от Finalize (), метод IDisposable.Dispose детерминирован, вызывается вручную и выполняется без задержки. В методе Dispose можно ссылаться как на управляемые, так и на неуправляемые ресурсы. Библиотека NET автоматически вызывает Dispose, если реализован IDisposable.

Упаковки приводят к дополнительным сборкам мусора.

Используйте метода Finalize только при крайней необходимости.

Для отладки управляемой кучи использовать CLR Profiler.

Присваивание объекту значения null делает объект недостижимым.

Пишите код с учетом того, что произойти может все что угодно. Язык C# спроектирован с учетом требований упреждающего программирования – стиля кодирования, направленного на решение возможных проблем до того, как они возникнут.

Всегда задавайте максимальный уровень предупреждений компилятора.

В новых приложениях обрабатывайте предупреждения как ошибки. Выполняйте совместный анализ кода при добавлении кода в систему управления версиями и при развертывании продукта.

Тестирование разделяется на верификацию и валидацию.

Лучше рассматривать покрытие кода как косвенное подтверждение надежности приложения. Покрытие функций – выполняется ли конкретная функция и если да, то как часто. Покрытие операторов – выполнялась ли строка кода и если да, то как часто. Покрытие путей – выполнялась ли определенная ветвь кода. Уровень покрытия кода должен быть определен прежде, чем начнется программирование, согласован и доведен до всех участников команды.

Если исходный код не удовлетворяет заданному стандарту, его нельзя передавать в систему управления версиями.

Комментируйте все и всегда. Комментарии должны почти надоедать.

Код не должен содержать трюков.

/// – комментарий документации.

В начале функции проверяйте корректность переданных в нее параметров.

По возможности решайте проблемы с помощью обработки ошибок, а не исключений.

По возможности функции должны возвращать значения, отличные от void.

Всегда проверяйте значение, возвращаемое функцией.

Не пишите небезопасный код.

Не перехватывайте «избранные» исключения.

Избегайте определения или использования объектных типов (boxing и downcasting снижают производительность).

Всегда используйте общие (generic) коллекции вместо стандартных.

По возможности вместо коллекций используйте массивы.

Не используйте собственные API-приложения для реализации функциональности, имеющейся в NET Framework Class Library (FCL).

Из всех операторов циклов отдавайте предпочтение циклу for (дисциплинирует).

Присваивайте неиспользуемым объектам null.

Если у вас есть выбор, всегда используйте блок вместо одиночного выражения.

Разбивайте сложное выражение на простые подвыражения.

Придерживайтесь рекомендаций CLS.

Проверяйте на overflow и underflow с помощью MinValue, MaxValue.

Избегайте ненужной рекурсии.

microsoft.com/patterns.

Шаблоны проектирования представляют собой известные и проверенные решения типовых задач программирования.

В плане разработки выделять время на отладку.

Разработчики оперируют разрабатываемой версией, а клиенты получают рабочую версию.

Основная причина возникновения проблем с ПО – недостаток образования.

watin.sourceforge.net.

Многопоточковые приложения тестировать на разном количестве ядер.

ADPlus.

Отладочные версии приложений не оптимизируются.

В отладочной версии веб-приложения не истекает срок действия запросов ASP.NET.

Для отладочных версий веб-приложений не предусмотрена пакетная компиляция.

Большое количество сборок приводит к фрагментации виртуальной памяти.

В отладочной версии веб-приложения клиентские библиотеки кода и статические изображения из файла `webresources.axd` не кешируются.

ILDASM/ILASM, Reflector, Son of Strike (SOS), Windbg, GFlags.

Трассировка = инструментирование.

Первичное исключение не выбрасывается, а передается на обработку приложению, в отличие от вторичного.

Перехват необрабатываемых ошибок на уровне страницы более приоритетен, чем их перехват на уровне приложения.

Полезные высказывания из книги «Правила разработки программного обеспечения» Маккарти

В разделе приведены цитаты из [6].

Разработка качественного ПО концептуально:

- изучить пользователей и позиционировать себя на рынке;
- определить спецификацию продукта, который соответствует запросам рынка;
- разработать ПО, выпустить его, занять достойное место в ряду основных производителей.

Запаздывание – обычное явление в разработке ПО.

Истинная задача управления разработкой ПО – привлечь как можно больше интеллекта и вкладывать его во все сферы деятельности, связанные с созданием программного продукта с применением всех дисциплин.

Интеллект проявляется в качествах: – творчество;

- ум;
- рассудительность;
- эффективность;
- четкость.

Самое трудное в процессе разработки ПО – добиться действительного участия создателей.

Основные аспекты деятельности в разработке:

- как заставить человека думать;
- о чем люди должны думать;
- как сделать человеческие мысли эффективными.

Поставка качественного ПО в срок просто дело применения здравого смысла.

Программист – ювелир.

Поощрять умственную деятельность.

Устранять причины, а не людей.

Большинство ресурсов должно уходить на интеллектуальную деятельность, а не в механические усилия.

Первостепенная обязанность руководителя – вовлечь каждого работника в процесс и держать его в этом процессе на протяжении всего проекта.

Программный проект – это процесс, проходящий 4 основные стадии:

- начальные шаги – создание разумной маркетинговой стратегии, проектирование продукта, создание клана разработки и первые шаги по

разработке (собрать работоспособную команду из хаоса, который царит в фирме);

– середина игры – длительный промежуток времени, начинающийся с первого нарушения графика разработки и оканчивающийся наступлением режима выпуска;

– режим выпуска – время выпуска продукта;

– запуск – время, когда продукт считается выпущенным.

Области действий хорошей разработки: – организация;

– конкуренция;

– клиент;

– проектирование;

– разработка.

Коллектив разработчиков выполняет следующие функции:

– управление проектом: создавать план разработки, вести внешние связи, снабжать необходимыми ресурсами, участвовать в проектировании;

– контроль качества: оценивать качество продукта, участвовать в проектировании;

– разработка: писать программный код, отлаживать программы, участвовать в проектировании;

– управление продуктом, маркетинг: создавать план выпуска сообщений и продукта, осуществлять связь с клиентом, участвовать в разработке;

– документация и обучение пользователей: систематизировать информацию, необходимую для использования продукта, участвовать в проектировании.

Главная функция контроля качества – постоянно оценивать состояние продукта, чтобы правильно сфокусировать деятельность всех членов команды.

Состояние продукта = тестирование и анализ.

Фирма не должна перемешивать желаемое с действительным.

Не должно быть борьбы за официально установленные полномочия.

Цель проектирования продукта в том, чтобы в его основу легли лучшие идеи.

Все споры о том, что лучше, должны быть решены до начала разработки.

Ссоры – выражение проблем, которые возникли в команде.

Распределять роли до начала разработки.

Для поддержания работоспособной, слаженной команды необходимо общее видение.

Для достижения равновесия нужно действовать в более узких рамках, специализироваться и конкурировать.

Каждый член команды должен знать:

- что будет делать команда;
- как будет выглядеть конечный продукт;
- какова базовая стратегия создания продукта;
- когда команда должна выпустить продукт, чтобы получить от него

ожидаемый эффект.

Все возникшие противоречия должны быть разрешены.

Хорошие лидеры любого уровня должны передавать свое видение приверженцам.

Установить в команде режим сопереживания.

Что делали бы эти люди на моем месте?

Как преобразовать их сложные чувства в ясный приказ о действиях?

Привлекать каждую голову в дело.

Если каждый член команды постоянно думает, какое поведение эффективнее всего в данном случае, то есть вероятность, что дела команды пойдут лучше.

Выслушивать все идеи.

Чем больше идей, тем лучше.

Идеи людей должны применяться.

Создать технологический план выпуска версий.

Технологический план – обновляемый 1–2 раза в год контракт, выражающий намерения команды.

Не реализовывать все задумки именно в этой версии.

Для отслеживания инвестиций и трудностей делить план на стратегические, конкурентные, ориентированные на клиента, инвестиционные и парадигматические отличительные особенности.

Когда кажется, что кто-то сумасшедший, тупой или убогий, то обычно следует более тщательно изучить человека, а не списывать его.

Обеспечить адекватное восприятие критики.

Правильно балансировать нагрузки в проекте.

Исключить перегорание разработчиков.

Провести разведку перед началом нового проекта, особенно для требований.

Использовать последние версии инструментов.

Прогнозировать версии сред, в которых будет эксплуатироваться продукт.

Необходимо соблюдать пропорции в численности команды по ролям.

У каждого одна своя роль.

Группа автора:

- 6 разработчиков;
- 2–3 контролера качества;
- 1 руководитель проекта;
- 2 технических писателя.

Контролер качества должен проектировать продукт и знать потребителей.

Можно добавить в команду маркетологов.

Управление областями по отдельным дисциплинам поручать ведущим специалистам в области.

Команда должна нести совместную ответственность за все аспекты.

Жест обратной связи гасит агрессию.

Необходимый команде консенсус достигается ценой бессонных ночей, страха быть отвергнутым, испытаний личной храбрости.

Члены команды приходят к выводу, что могут действовать свободно и несут ответственность за свои действия.

Не должно быть поддакивания, фальшивого консенсуса.

Члены команды должны осознать, что им никто не мешает.

Каждого человека надо вдохновлять и подпитывать.

Поначалу становлению команды могут препятствовать ее члены.

В идеальном проекте имеются создатели – специализируются в разработке, контроле качества, маркетинге, документировании и так далее, и кураторы – специализируются в понимании группы, создании обстановки для творческой работы и эффективной реализации идей.

Кураторы обеспечивают включение максимального количества идей в коробку с программным продуктом.

Кураторы и создатели ответственны за качество и состояние группы.

Единственная власть идет от знания и понимания, а не от занимаемой позиции.

Руководители проекта выполняют функции:

- возглавляют определение успешного продукта;
- возглавляют распространение видения продукта;
- возглавляют движение команды к гарантированной поставке продукта.

Руководитель проекта не имеет прямой власти.

Роль руководителя проекта – лидерство.

Управление проектом – техническая задача.

2 аспекта технического мастерства: технология, используемая при создании ПО, продукта, и технические аспекты лидерства при создании ПО.

Руководитель проекта должен уметь уговаривать, содействовать, внушать вдохновение, требовать совершенства, эффективной работы от остальных членов команды, быть представителем для прессы, клиентов и руководства корпорации.

Руководитель проекта – сердце разработки ПО.

ПО выражает команду, его создавшую.

Следить не за словами и поведением команды, а оценивать ее по ПО.

Анализ проблем в команде – способ привести ПО в порядок.

Постоянно поддерживать контакт с духом команды.

Сложные чувства и идеи должны свободно передаваться внутри команды.

Сделать максимально полной властью каждого члена команды.

В команде не должно быть авторитетов и кумиров.

Вводить команду в курс дела, чтобы сформировать адекватные ожидания.

Менеджеры обеспечивают принятие командой хорошего решения – обучением, информацией, адекватными ресурсами любого вида.

В среде с правильным наделением полномочиями ситуация не анархическая и конфронтационная, а основанная на способностях людей.

Сфокусировать внимание команды на выпуске продукта.

Игнорировать иерархические и функциональные границы.

Задача создания ПО не имеет отношения к тому, кто и какие принимает решения, кто и за что вознаграждает, кто и за что наказывает.

Четыре основные ситуации на рынке: – вы один на рынке;

– вы идете бок о бок с конкурентом;

– вы отстаєте от конкурента;

– вы опережаете конкурента.

Необходимо знать, в каком направлении развивается рынок, кто конкуренты, каковы их возможности, как их опередить.

Человек генетически предрасположен к командной работе.

Люди в вашей группе были отобраны эволюционно.

Как только поставлен диагноз ситуации, сразу обсудить классические ходы.

Рынка без конкуренции не бывает.

Комплекс задач: создание отличительных преимуществ, разработка технологического плана.

Перегруженный функциями и отличительными особенностями продукт становится раздутым и ненадежным.

Силовой подход: опередить конкурента по числу отличительных особенностей или добавить парадигматические особенности, меняющие архитектуру.

При неудаче сохранить расходы на пиар.

Выпустить раньше, чем это сделают конкуренты.

При лидерстве самодовольство – враг.

Лидерство не будет вечно.

Надо конкурировать с самим собой.

Работать над увеличением скорости перемен.

Пресекать все попытки возврата к худшему.

Не позволять людям успокаиваться и замедлять движение.

Цикл ПО связан с выпусками ОС.

Проектировать желаемость продукта с самого начала.

Принимать во внимание психоэмоциональное состояние клиента, в котором он использует компонент.

Нормальная успешная фирма в версии за версией отвечает на самые жесткие требования клиентов.

Рынок хочет делать на компьютерах то, что раньше на них не делалось.

Спрашивать клиентов об их нуждах всегда и везде.

Проводить статистические исследования.

Всегда думать о власти, безопасности и управлении.

Не будьте непостоянны с вашими клиентами.

Сокращать время цикла разработки.

Наличные деньги никогда не лгут.

Клиент должен быть в состоянии прогнозировать сроки и возможный эффект следующей версии.

Клиенты предпочитают новое ПО обещаниям.

Проанализируйте значение вашей работы в исторических терминах.

В ПО каждый элемент существенен для целого и все существенные элементы присутствуют.

Пользователю ничего, кроме продукта, не нужно.

Тема (theme) ПО – преобладающая идея, создающая основу для проектирования.

Определив тему, нужно жертвовать другими качествами.

Элементы продукта получают внимание пропорционально их важности.

Ранние части ПО определяют более поздние.

Минимизировать зависимости.

Исключить самую вероятную опасность.

Тщательно выбирать платформу.

Устанавливать сроки на стадии проектирования.

Разработка = планирование, составление графиков, создание и развертывание ПО.

Создавать контрольные точки.

Не слушать некомпетентных приказов.

В сумасшедшем доме здоровый человек считается сумасшедшим.

Не позволять абсурду и неразумности укреплять позиции.

В середине игры цели – стойкость и упорство.

Некоторый риск существует даже в простых процедурах.

Не поддаваться страхам команды.

Треугольник: отличительные особенности, ресурсы, время.

Запросивший помощь должен максимизировать эффективность количества и типа помощи. В таких случаях ничего нельзя сказать наверняка.

Явно озвучивать ваше незнание, чтобы все понимали истинное положение вещей.

Вести списки относящихся к делу аспектов, которые на данный момент неизвестны.

Предпосылками успеха являются лишения и разочарования.

Не замалчивать неопределенности.

Равновесие, созгласованность, целостность.

Мысли = слова = дела.

Сдаваемые объекты должны появляться через небольшие промежутки времени.

В разработке запаздывали все.

Задача разработки – формирование группы, способной создать и выполнить сотни маленьких, но существенных обязательств.

Не оставлять разработчиков наедине с собой.

Продукт следует собирать с незначительными интервалами.

Не уходить в темноту.

Функциональность системы не должна теряться.

Текущая сборка – это текущее состояние команды.

Достигнуть состояния ясности и оставаться в нем.

Иметь выпускаемый продукт каждый день.

Нужен человек, занимающийся формулированием состояния дел.

Использовать метрики каждый день.

Намечать контрольные точки для каждой, даже самой маленькой задачи.

Для контрольных точек назначать уровень качества.

Команде необходимо осознавать свое состояние.

Цели контрольной точки выражаются в терминах объектов сдачи.

Объекты сдачи передаются с письменным подтверждением заинтересованных лиц.

В команде производить балансировку нагрузки.

Каждую контрольную точку обсуждать без поиска виновных.

Выдвинуть на первый план моменты, которые были сделаны хорошо.

Не наказывать за опоздание в контрольной точке.

Контрольная точка: что будет передано, кому, в какой форме?

В команде – свобода высказываний.

К каждой контрольной точке выработать норматив.

Пренебречь элементами, представляющими избыточную сложность.

Контрольные точки от 6 недель до 3 месяцев друг от друга.

В проекте не более 7 контрольных точек.

К каждой контрольной точке формулируется 1–2 предложения, описывающих путь достижения.

Обсуждать с командой ситуацию и разрабатывать согласованный план действий.

Каждый новый проект начинается на нулевом уровне реальности.

Опоздания и ошибки не имеют отношения к проигрышу или вине.

Делать допуски в определении дат.

Не кричать о пугающем состоянии дел часто.

Что говорит о себе команда своим поведением?

Не искать козла отпущения.

Не переносить даты.

После опоздания следующая контрольная точка должна быть достигнута.

Делать опоздание положительным переживанием.

Наблюдать за всем.

Не все изменения следует принимать и реализовывать.

Анализировать причины и цели проблем.

Изменяться вместе с миром.

Не принуждать людей работать по вечерам и выходным.

Нарушать правила – разработчики любят вставать против власти и структуры.

Применять символическое общение.

Бета-версии должны обеспечить проверку работы продукта на как можно большем числе машинных конфигураций.

Никакие реальные изменения, за исключением исправлений конфигурационных проблем, не вносятся, а переходят в следующую версию.

У маркетологов должна быть эмоциональная модель клиентов.

Критерии сортировки дефектов: серьезность дефекта, важность, размер, возможность дестабилизации ПО при исправлении дефекта, динамика команды, степень полноты тестирования продукта после исправления дефекта.

Качество включает в себя новизну проекта и его исключительность.

Фанфары – важная часть отношений с клиентом.

Все события, предшествующие событию запуска и следующие за ним, должны быть строго расписаны.

История продукта должна быть оригинальной житейской мудростью.

У людей нет ни времени, ни желания овладевать сложными вопросами.

Хороший рассказ – настроение, замысел, хорошие и плохие парни, накал страстей и сцены погони.

1–2 ключевых фактов достаточно.

Сделать так, чтобы слушатель выводил сам главную вещь, но не сообщать ее.

Мода – мощный мотивирующий фактор.

Пользователи идентифицируют себя с ПО.

Член команды должен быть прежде всего сообразительным.

Неповторимость – наиболее видимое проявление интеллекта.

Ставить кандидатам реальные задачи.

Дать людям команды перспективу.

Здоровые люди от голода не умирают.

Все новое встречает противодействие.

Почти всегда легче использовать сильные стороны человека, чем исправлять его недостатки.

Не бойтесь показать свою уязвимость.

Использовать более функциональную модель для регулирования отношений между боссами и подчиненными.

Босс – клиент, разработчик – поставщик услуг.

Если босс собирается сделать глупость, то не реагировать.

Оказание наилучших услуг не граничит с самоунижением.

Когда в группе отсутствует функциональный порядок или иерархия, в группе всегда присутствует борьба (скрытая или явная) за позиции и связанная альфа-энергия.

Применять атаку с поклоном.

Вы не насладитесь удобствами и особенностями здания до тех пор, пока не поселитесь в нем.

Удаленность – центральная проблема в коллективе.

Свободное выражение эмоций в группе.

Каждый должен знать, чего он хочет и чего от него хотят.

Полезные высказывания из книги «Увеличение продаж с SEO» Дыкана

В разделе приведены цитаты из [7].

Строить таблицу из трех колонок: пример запроса пользователя, что пользователь хочет увидеть на странице, что мы размещаем на странице, дабы угодить пользователю.

Таблицу из двух колонок: сотрудники, отвечающие за сайт.

Теги в порядке влияния: title, h1, h2...

Ссылки заголовками не обозначают.

H1 – единственный главный заголовок на странице, остальные Hn – подзаголовки.

Оптимальный вариант – количество тегов Hn на странице не превосходит их номера n.

Лучше не заниматься созданием «лишних» заголовков.

При написании заголовков употребление ключей желательно, но не должно раздражать человека.

Заголовки должны быть приятны уху и глазу, вызывать приятные эмоции.

Уникальный текст страниц сайта важен для поисковика.

Должно быть (или используются) немного (!) точных цитат, имен собственных, популярных расхожих фраз (афоризмов, пословиц, поговорок и тому подобного).

Рерайт – переписывание чужого текста своими словами.

Проверка на уникальность текста: text.ru, Advego Plagiatius.

Большинство заказчиков готовы принять текст с уникальностью 95–97%.

Качественный текст – это грамотно изложенный, красиво написанный, верно структурированный текст без синтаксических ошибок.

Для поисковиков важна и достоверность изложенной информации.

Поисковики не слишком одобряют дублированный (повторяющийся) контент на одном ресурсе (сайте).

Высоко котируются тексты, не содержащие орфографических и пунктуационных ошибок, опечаток, помарок и недописок.

Все слова должны быть прописаны максимально полно (без тчк, зпт).

Ключевое слово – слово или словосочетание, которое теоретически пользователь введет в поисковик.

Ключ в прямом вхождении – фраза в неизменном виде, которую вводят в строку поиска и которая встречается на сайте.

Бывают ключи непрямого вхождения – слова ключа разделяются другими словами.

Весь перечень ключевых слов и словосочетаний, имеющих в тексте, называется семантическим ядром.

В семантическое ядро также входят слова, которые, возможно, и не являются ключами, но вводятся пользователями в строку поиска вместе с ключевыми запросами, а также определенные формы поисковых слов.

Семантическое ядро демонстрирует тематику сайта.

Ключевые слова не должны повторяться на сайте слишком часто.

Оптимальная длина текста – от 1000 до 4000 знаков.

Важный фактор – количество кликов по ссылке на сайт в выдаче поисковика.

Уход с сайта после захода не нравится поисковику.

Релевантность страницы запросу пользователя – уровень соответствия страницы или сайта ожиданиям пользователя (по ней определяется позиция и наличие ссылки в выдаче).

Текст должен быть адекватным, интересным, легко читаемым, тематическим и без «воды».

Поисковая система уточняет, в прямом ли вхождении встречаются все слова (фразы) запроса в тексте или между ними есть еще какие-то слова (то есть фраза запроса разделена другими словами – от этого зависит позиция в выдаче).

Чем меньше расстояние (побуквенное) между словами запроса в тексте сайта, тем более релевантной считается страница.

Чем ближе морфологическая форма слов в тексте к той морфологической форме, которая указана в запросе, тем более релевантным считается текст (падеж, число).

Поисковик обращает внимание на синонимы, количество слов или словосочетаний запроса, присутствующее в тексте.

Наполнять сайт бесполезной для пользователя информацией нежелательно.

Для поисковика важны время, проведенное на сайте, и количество просмотренных страниц.

Характеристики хорошего сайта:

- сайт прост и понятен;
- дизайн пришелся по душе пользователю;
- контекст сайта – исключительно нужная и полезная информация;
- пользователь нашел, где писать;
- пользователь сумел найти необходимые ему контакты;
- пользователь поверил всему, что написано на сайте;
- пользователь узнал много нового на сайте (избегайте баянов).

Ссылочное ранжирование – это ссылки на ваш сайт, размещенные на других ресурсах.

Важен текст ссылки (анкор), по которому поисковик определяет, чем полезен сайт, на который ведет ссылка.

Важны авторитет разместившего ссылку, тематика сайта, на котором ее разместили, возраст ссылки.

Важна структура сайта: поисковая система должна верно понять содержание ресурса и проиндексировать его.

Важно наличие иерархической структуры сайта.

Желательно, чтобы пользователь добрался до нужного ему раздела за три клика (максимум – четыре).

Перелинковка – внутренние ссылки, установленные с одной страницы сайта на другую.

Страница, на которую ставится ссылка, имеет при прочих равных условиях больший вес, чем страница, на которую ссылка не стоит.

Анкоры в перелинковке используют для продвижения по низкочастотным запросам.

В анкоре должен быть прописан ключ продвигаемой страницы.

Скорость загрузки – важный фактор, влияющий на конверсию и ранжирование.

Количество языковых ошибок на сайте проверяется поисковиком.

Поисковая система обращает внимание на географический фактор.

Для поисковика имеет значение количество упоминаний о вашем сайте в соцсетях, ссылки на ваш сайт, размещенные в соцсетях, и присутствие вашего сайта в соцсетях.

SMM – Social Media Marketing.

Для поисковика важен возраст сайта.

Чужие тексты, картинки, графики кардинально снижают рейтинг сайта.

Допустимы только рерайт – видоизмененные первоисточники, и копирайт – написание текста с нуля без первоисточника.

Ссылки с сайтов, которые были созданы для продажи ссылок, снижают рейтинг в выдаче.

Все разделы сайта должны быть расположены в соответствии с логикой и степенью значимости.

При правильной структуре больше веса должно приходиться на продвигаемые страницы (ненужные расположены в недрах сайта).

Нужно учитывать эффективность ключей.

Что еще можно сделать для продвижения сайта?

Как можно сделать сайт еще удобнее?

Пользователю важен шриффт.

Делайте, как для себя.

Навигация сайта должна быть максимально легкой и удобной: зашел, увидел, выяснил, обрадовался, купил.

Сайт должен быть приятным глазу.

Вы должны создать картинку, при виде которой у пользователя поднимется настроение.

Сайт должен быть аккуратным, иметь свой стиль, лаконичным.

Красивый сайт выглядит лучше конкурентов из топа тематики.

Собрать семантическое ядро означает подобрать список ключей, который максимально полно будет включать в себя все товары, услуги и предложения, которые представлены на сайте заказчика.

Для этого нужно узнать, как именно ищут эти товары и услуги (какие слова и словосочетания вводят в строку поиска).

«Яндекс. Директ».

При составлении семантического ядра нужно стремиться к тому, чтобы использовать наименования всех услуг и товаров, предлагаемых на сайте, уделяя внимание каждой услуге и каждому товару, так как нет гарантии вывода в топ конкретного ключевого слова или словосочетания, поэтому необходимо расширение списка ключей.

Для того чтобы выдвинуться в топ по самым популярным запросам, сайт должен быть не хуже тех, которые уже представлены в топе.

«Яндекс. Вордстат» – общая частота запроса включает все запросы, которые в него входят (если он введен в «Вордстат» без кавычек, с кавычками – разное совпадение формулировки запроса).

Ключи могут быть раскручены мошенниками.

Каждая статья должна быть уникальной, информативной, интересной.

Составлять таблицу из двух столбцов: запрос, цель пользователя.

Выбор – это привилегия тех, кто платит.

Лучшие акции нужно искать у конкурентов.

Сайт с отзывами вызывает у пользователей больше доверия.

На сайте нужны бонусы, скидки, акции, заманчивые предложения.

Один час сегодня стоит двух часов завтра.

Нужно использовать онлайн-консультант.

С помощью рассылок можно возвращать пользователей на сайт.

Призыв к действию должен быть четким и понятным.

Дата проведения акции должна быть адекватной.

Для заказа только по номеру телефона делайте кнопку быстрого заказа, а также кнопку «заказать обратный звонок».

Предлагайте подарки к покупкам.

Заголовок каждой страницы лучше включать в TITLE.

Во всех уникальных текстах страницы должны быть продвигаемые фразы.

Уделяйте внимание разметке текстов.

copyscape.com.

Сайт с дублированным контентом поисковик считает недостойным внимания пользователя.

Страховка от воровства уникальных текстов – в «Яндекс. Веб-мастер».

При потере позиций на сайте нужно обновлять контент на уникальный.

Страницы, продвигающие более конкретные запросы, должны быть глубже в структуре сайта.

«Все, что может быть сказано, может быть сказано ясно».

Структура сайта должна быть правильной и понятной; товар должен быть в широком ассортименте; описывающие тексты должны быть уникальными.

«Чем больше слов произносишь, тем туманнее мысль».

Видеофайлы и картинки могут помочь задержать пользователя на сайте, что важно для поисковых систем.

Около 90% сайтов требуют переделки под оптимизацию.

Оценить количество ссылок, стоящих на сайты-конкуренты: linkpad.ru.

Желательно, чтобы на сайте не было рекламы.

Создание семантического ядра займет 3–4 недели.

Добавить сайт в справочник адресов «Яндекс» и в «Яндекс. Каталог».

Примерно месяц поисковик будет замечать обновление контента.

Скупой платит дважды.

Пользователь должен мочь почерпнуть из сайта то, что ему пригодится.

Можно гарантировать действия, но нельзя – результат.

SEO-оптимизатор – главный человек в команде.

Лучше, когда продвигается большое число ключей.

«Гораздо лучше обещать меньше и давать больше, чем наоборот».
searchengines.ru.

Если ресурс резко просел, то либо поисковик поменял критерии, либо на сайте что-то испортили.

Полезные высказывания из книги «Факты и заблуждения профессионального программирования» Гласса

В разделе приведены цитаты из [8].

Хороший менеджмент важнее хорошей технологии.

Те специалисты, которые действительно хороши в том, что они делают, и в то же время находятся на дне служебной иерархии, обладают возможностью, которой не имеет никто другой в этой пирамиде: их нельзя понизить в должности – в этой позиции скрыта большая сила.

Люди важнее, чем средства, методы и процесс.

Для пользователя важны высокие умственные способности и одержимость разработчика желанием заставить программу работать именно так, как она должна, – все остальное вторично.

Лучшие программисты до 28 раз превосходят слабейших, но оплата их труда не бывает соразмерной, поэтому лучший программист – самое выгодное приобретение в индустрии ПО.

Корреляция между результатами тестов и показателями на рабочем месте равна нулю.

Корреляция между оценками по информатике и производительностью труда тоже ужасна.

Если проект не укладывается в сроки, то добавление рабочей силы задержит его еще больше.

Разработка ПО – это интенсивная умственная деятельность, и среда, в которой она проходит, должна способствовать мышлению.

То, что можно надежно измерить («твердое»), обладает свойствами перетягивать внимание от того, что нельзя точно измерить (от «мягкого»).

Рекламный звон вокруг инструментов и методов («дешевле, лучше быстрее», «новое лучше старого») – это чума индустрии ПО, а реальное увеличение производительности и качества – от 5% до 35%.

Изучение нового метода или средства сначала понижает производительность программистов и качество продукта (см. кривую обучения – learning curve).

Назовите любую пропагандируемую концепцию, и найдется кто-то, кто будет утверждать, что проекты становятся неуправляемыми именно потому, что она мало применяется.

«Оптимистичная оценка» является главной причиной нарушения сроков выполнения в 51% проектов.

Сначала требования, потом – оценка.

Можно ли оценить время и затраты на решение задачи, не имея о ней представления?

70% оценок делают кем-то, кто связан с отделом по работе с пользователями, а 4% приходится на проектную команду.

Оценки в проектах ПО желательно корректировать впоследствии.

Правильно оценивать никто не будет, поэтому нет причин беспокоиться о том, что проекты ПО не завершаются в сроки, задаваемые оценками.

В нашей современной культуре принято во что бы то ни стало укладываться в сроки (невозможные), жертвуя ради этого завершенностью и качеством.

Создание полностью универсального компонента для повторного использования в крупном масштабе – сложная задача.

Универсальные решения требуют в разы больше времени.

Модификация повторно используемого кода крайне чревата ошибками.

Решение модифицировать пакетную программную систему от стороннего производителя практически всегда ошибочно.

Если код ПО предстоит модифицировать на 20–25% или больше, то проще и дешевле начать все заново и создать новый продукт.

Разработка ПО – это деятельность на 80% интеллектуальная и на 20% техническая.

Требования должны меняться в процессе.

Важно отказаться от обреченных на неудачу попыток создать ПО без ошибок, чтобы сконцентрироваться на более реалистичных и достижимых целях.

Нужно применять анализаторы тестового покрытия.

За час можно сделать ревью примерно 100 строк кода.

Инспекции могут приводить к спорам, ухудшающим моральный дух команды.

Стоимость сопровождения в среднем 60% стоимости ПО, из них 60% – на модернизацию.

Сопровождение более трудоемко, чем разработка.

Жизненный цикл разработки: 20% – требования, 20% – проектирование, 20% – кодирование, 40% – устранение ошибок.

Улучшение качества разработки ПО приводит к тому, что сопровождения становится больше из-за облегчения внесения большого количества изменений в связи с применением обновленного инструментария и подходов.

Качество ПО есть совокупность свойств:

1. Переносимость означает, что программный продукт можно без труда перенести на другую платформу.

2. Надежность – это свойство программного продукта надлежащим образом выполнять свои функции.

3. Под эффективностью программного продукта понимают экономное расходование им времени и занимаемого места.

4. Принятие в расчет человеческого фактора (что называют также словом «юзабилити») подразумевает, что с программным продуктом легко и удобно работать.

5. Тестируемость ПО есть свойство, характеризующее легкость его тестирования.

6. Понятность ПО – это свойство, характеризующее, насколько легко (или трудно) специалисту, сопровождающему программный продукт, понять его работу;

7. Модифицируемым называют ПО, изменение которого не вызывает трудностей.

Удовлетворение пользователя = выполнение требований + своевременная поставка + приемлемая стоимость + качественный продукт.

Эффективность кода на ЯВУ, скомпилированного с оптимизацией, может достигать более 90% аналогичного ассемблерного кода (спор решен в 70-х).

Невозможно управлять тем, что невозможно измерить, – заблуждение.

В обучении программированию важно учить сначала читать код.

Полезные высказывания из книги «Путь программиста» Сонмеза

В разделе приведены цитаты из [9].

Вы получите все, что хотите в жизни, если достаточно поможете другим людям.

Надо быть не успешным, а ценным.

Вы получите ровно столько, сколько вкладываете.

Человек, имеющий наибольший интерес, всегда находится в невыгодном положении на переговорах.

Большинство людей думают, что деньги имеют значение только в краткосрочной перспективе.

Большой заработок не делает человека финансово образованным.

Актив приносит больше, чем стоит, пассив – меньше.

Ваши убеждения становятся вашими мыслями.

Ваши мысли становятся вашими словами.

Ваши слова становятся вашими действиями.

Ваши действия становятся вашими привычками.

Ваши привычки становятся вашими ценностями.

Ваши ценности становятся вашей судьбой (Махатма Ганди).

Упади семь раз, встань восемь (японская пословица).

Ни у кого нет монополии на правду.

Написание книги «Путь программиста» заняло 500 часов.

Все должны помогать друг другу.

Полезные высказывания из книги «Рефакторинг. Улучшение существующего кода» Фаулера

В разделе приведены цитаты из [10].

Рефакторинг (refactoring) – изменения во внутренней структуре программного обеспечения, имеющие целью облегчить понимание его работы и упростить модификацию, не затрагивая наблюдаемого поведения.

Производить рефакторинг (refactor) – изменять структуру программного обеспечения, применяя ряд рефакторингов, не затрагивая его поведения.

Рефакторинг предоставляет технологию приведения кода в порядок, осуществляемую в более эффективном и управляемом стиле.

При каждой возможности проводить тестирование.

Цель рефакторинга – упростить понимание и модификацию ПО.

Оптимизация производительности часто затрудняет понимание кода.

Рефакторинг не меняет видимого поведения ПО.

Рефакторинг улучшает композицию ПО.

Изменение размера образа программы в памяти редко имеет значение.

Удаление дублирующегося кода улучшает композицию.

Рефакторинг облегчает понимание ПО.

Рефакторинг делает код легким для чтения.

Рефакторинг позволяет увидеть в коде больший объем системы.

Рефакторинг помогает найти ошибки.

Рефакторинг ускоряет понимание программ.

Рефакторингом следует заниматься постоянно понемногу.

Сначала необходимость рефакторинга будет неприятна.

Чаще всего рефакторинг начинается с необходимости добавления в ПО новой функции.

Рефакторинг – процесс быстрый и ровный.

Получение сообщения об ошибке – причина провести рефакторинг.

Проводить рефакторинг при разборе кода.

Рефакторинг облегчает модификацию.

Рефакторинг способствует получению более конкретных результатов от разбора кода.

Использовать UML и CRC-карты при разборе дизайна кода.

Рефакторинг разделяет большие объекты на несколько меньших.

Благодаря помещению отдельного программного слоя между объектной моделью и моделью базы данных можно отделить модификации двух разных моделей друг от друга.

Если изменяемый интерфейс используется недоступным для изменения кодом, то из старого интерфейса вызывать новый, пометив старый как устаревший.

Не нужно предоставлять интерфейсы, которые не требуются.

Не публиковать интерфейсы раньше срока.

Определять родительский класс исключения для пакета в целом.

Представлять себе возможный рефакторинг.

Перед началом рефакторинга код должен выполняться в основном корректно, иначе – переписать код с нуля.

Рефакторинг должен подразумевать его завершение.

Окончание проекта может быть причиной отложения рефакторинга.

Программа весьма отличается от физического механизма.

Рефакторинг может быть полной заменой предварительному проектированию.

При рефакторинге от предварительного проектирования требуется приемлемое решение, а не единственно правильное.

Гибкие решения сложнее обычных.

Даже если точно известно устройство системы, не заниматься гаданием, а провести замеры.

Секрет создания быстрых программ: написать настраиваемую программу, а затем настроить так, чтобы достичь приемлемой скорости.

Подходы написания быстрых программ:

- при декомпозиции каждому компоненту выделяется бюджет ресурсов – по времени и памяти; компонент не выходит за рамки бюджета, хотя разрешен обмен ресурсами;

- постоянное внимание и попытки программиста сделать систему производительнее.

Если в равной мере оптимизировать весь код, то 90% оптимизации будет произведено впустую.

Большая часть времени расходуется небольшой частью кода.

Запуск в конце разработки профайлера и выявление затратных компонентов.

Рефакторинг выигрывает время для оптимизации.

Рефакторинг обеспечивает более высокое разрешение для анализа производительности.

Рефакторинг возник в 80-х.

Одно и то же выражение в двух методах одного класса – необходимо выделение метода.

Одно и то же выражение в подклассах одного уровня – необходимо выделение метода с последующим подъемом поля.

Если код похож, но не совпадает полностью, нужно применить выделение метода для отделения совпадающих фрагментов от различающихся, затем проверить применимость формирования шаблона метода.

Если два метода делают одно и то же с помощью разных алгоритмов, можно выбрать более четкий из алгоритмов и применить замещение алгоритма.

Если дублирующийся код находится в двух разных классах, попробовать применить выделение классов в одном классе, а затем использовать новый компонент в другом.

Следует активнее применять декомпозицию методов.

В 99% случаев, чтобы укоротить метод, требуется лишь выделение метода.

Устранить временные переменные метода заменой временной переменной вызовом метода.

Длинные списки параметров метода сокращать введением граничного объекта, сохранением всего объекта.

Если не удалось удалить все лишние временные переменные и параметры, применить замену метода объектом метода.

Даже одну строку имеет смысл выделить в метод, если она нуждается в комментарии.

Для условных выражений делать выделение с помощью декомпозиции условных операторов.

Содержащийся в цикле код – отдельный метод.

При большом числе атрибутов класса применять выделение класса.

Одинаковые префиксы или суффиксы подмножества переменных – необходимость создания компонента.

Для создания компонента как подкласса использовать выделение подкласса.

Если класс не использует постоянно все переменные своего экземпляра – применить выделение класса и выделение подкласса несколько раз.

Для класса с чрезмерным объемом кода применить выделение класса или выделение подкласса, а для дальнейшего разделения – применить выделение интерфейса и выявить еще части.

При необходимости хранения копий некоторых данных в двух местах и обеспечения их согласованности применить дублирование видимых данных.

Класс GUI выделить с его данными и поведением в отдельный объект предметной области.

Работая с объектами, следует передавать методу не все, а столько, чтобы он мог добраться до всех необходимых ему данных.

Для получения данных в одном параметре путем вызова метода применить замену параметра вызовом метода объекта, который уже известен.

Чтобы группу данных, полученных от объекта, заменить самим объектом, применить сохранение всего объекта.

Если есть несколько элементов данных без логического объекта, применить введение граничного объекта.

Если не нужно создавать зависимость между вызываемым и крупным объектом, можно передавать все в параметрах.

Если класс часто модифицируется различными способами по разным причинам, необходимо разбиение на классы.

Определить, что изменяется по одной причине в классе, и применить выделение класса.

Когда при выполнении любых модификаций приходится модифицировать множество классов с небольшими изменениями, применить перемещение метода, перемещение поля, чтобы свести все изменения в один класс.

Чтобы поместить связку методов в один класс, использовать встраивание класса.

Если метод больше интересуется не тем классом, в котором он находится, применить перемещение метода или выделение метода для части кода этого метода.

Если метод использует функции нескольких классов, то разбить его выделением метода.

То, что изменяется одновременно, надо хранить в одном месте (для исключений – паттерны стратегия и посетитель).

Связки данных, встречающихся совместно, надо превращать в самостоятельный класс: сначала найти эти группы в виде полей, применить к ним выделение метода (в отдельный класс), затем к сигнатурам методов применить введение граничного объекта или сохранение всего объекта.

Элементарные типы убирать применением замены значения данных объектом или замену кода типа классом.

Для условных операторов, зависящих от кода типа, применять замену кода типа подклассами или замену кода типа состоянием, стратегией.

При наличии группы полей, которые должны находиться вместе, применять выделение класса.

Для примитивов в типах параметров применять введение граничного объекта.

Если обнаружена разборка на части массива, применить замену массива объектом.

Для выделения переключателя switch использовать выделение метода, затем перемещение метода, затем пробовать применить замену кода типа подклассами, замену кода типа состоянием или стратегией, замену условного оператора полиморфизмом.

Если есть лишь несколько вариантов переключателя, управляющих одним методом, и не предполагается их изменение, применить замену параметра явными методами.

Если одним из вариантов является null, применить введение объекта Null.

Чтобы заставить экземпляры одной иерархии ссылаться на экземпляры другой, применить перемещение метода и перемещение поля.

При наличии подклассов с недостаточными функциями применить свертывание иерархии, встраивание класса.

Если есть абстрактные классы, не приносящие большой пользы, применить сворачивание иерархии.

Ненужное делегирование можно устранить с помощью встраивания класса.

К методам с неиспользуемыми параметрами применить удаление параметров.

К методам со странными абстрактными именами применить переименование метода.

Для непонятных переменных применять выделение класса.

Условно выполняемый код удалить, применив введение объекта Null.

Для алгоритма, использующего несколько переменных, применить выделение класса.

Для цепочки сообщений применить сокрытие делегирования (для конечного объекта можно применить выделение метода, перемещение метода и передвинуть использующий его код вниз по цепочке).

Для интерфейса, в котором половина методов делегирует обработку другому классу, применить удаление посредника.

При наличии нескольких методов, не выполняющих большой работы, применить встраивание метода и поместить в вызывающий метод.

Посредника преобразовать в подкласс реального класса можно заменой делегирования наследованием.

С помощью перемещения метода и перемещения поля можно разделить части и уменьшить близость классов, затем применить замену двунаправленной связи однонаправленной, выделением класса, сокрытием делегирования.

При чрезмерном наследовании применить замену наследования делегированием.

Ко всем методам, выполняющим одинаковые действия по различающимся сигнатурам, применить переименование метода, перемещение метода, выделение родительского класса.

Если в библиотечный класс нужно включить 1–2 новых метода, то применить введение внешнего метода.

Если в библиотечный класс нужно включить много новых методов – применить введение локального расширения.

К открытым полям классов применить инкапсуляцию поля.

К открытым полям коллекций применить инкапсуляцию коллекций.

Ко всем полям, значение которых не должно изменяться, применить удаление метода установки значения.

Методы доступа переместить в класс данных, применив перемещение метода, выделение метода.

К методам получения и установки значений полей применить сокрытие метода.

При неправильной иерархии наследования применить спуск метода, спуск поля, создав новый класс на одном уровне с потомком, вытолкнуть в него бездействующие методы.

Можно делать все родительские классы абстрактными.

Для разрушения иерархии при неиспользовании интерфейса родительского класса применить замену наследования делегированием.

Чтобы избавиться от комментариев к блоку, использовать выделение метода, переименование метода, введение утверждения.

Перед рефакторингом должны быть созданы тесты.

В каждом классе должен быть свой метод, с помощью которого он может себя протестировать.

Тестовые данные – test fixture.

Проверять, действительно ли тест проверяет то, что требуется.

Получив сообщение об ошибке, начать с теста модуля, показывающего ее.

Тестировать те области, возможность ошибок в которых выше.

Проверять, что ожидаемые ошибки происходят в надлежащем порядке.

Проверять тестами границы.

Для представления денежных величин использовать паттерн количество.

Выделение метода и другие рефакторинги – смотреть по шагам в приложении книги.

Полезные высказывания из книги «Видимость в интернете» Тероу

В разделе приведены цитаты из [11].

Трафик (traffic) – количество посетителей, пришедших на веб-сайт за определенный промежуток времени (обычно за сутки).

Результаты поиска – Search engine results page (SERP).

По оценке 2006 года, среднестатистический пользователь за месяц просматривает примерно 93 страницы результатов поиска, в среднем затрачивая на поиск около 27 минут.

Основные задачи поисковых систем:

1. Поисковый робот – «паук» (spider), отыскивает страницы и включает в индекс (этот способ сбора называется сканированием – crawling, spidering).

2. Индексатор найденные роботом на доступных ему страницах слова и словосочетания помещает в индекс на сервере поисковой системы.

3. Процессор запросов, часть поисковой системы, сопоставляет ключевые слова и словосочетания из строки запроса с вебстраницами из индекса и отбирает страницы, наиболее близко соответствующие запросу пользователя.

Алгоритмы поисковых систем хранятся в строгом секрете и ежедневно изменяются.

Все, что мешает этому процессу, негативно скажется на позиции сайта в поисковых системах. Поисковые роботы:

1. Переходят по ссылке на веб-адрес (URL).

2. Запрашивают URL у вашего веб-сервера, ваш веб-сервер передает веб-страницу поисковой системе.

3. Составляют список слов и словосочетаний, найденных на странице с данным URL.

4. Определяют «вес», или релевантность, данных слов и словосочетаний на странице. Передают информацию в индекс поисковой системы.

5. Когда пользователи обращаются к поисковой службе, та выполняет поиск по индексу (обычно индекс обновляется каждые 2 недели).

Что делают эксперты по поисковой оптимизации:

1. Убеждаются, что ключевые слова и словосочетания занимают на ваших страницах стратегически важные места, вне зависимости от текущих алгоритмов поисковых систем.

2. Обеспечивают роботам свободный доступ к вашим страницам.

Фактически 95% добавлений, произведенных с помощью Add URL, являются спамом.

Наиболее приемлемый для большинства веб-страниц вариант уведомления о существовании сайта – обнаружение страницы поисковым роботом при штатном сканировании сети.

Поисковая система, применяющая модель оплаты за включение (pay-for-inclusion model), включает страницы веб-сайта в свой индекс за определенную плату.

Реклама в поисковых системах включает в себя спонсорство и плату за размещение (PFP – pay for placement).

Основные факторы успеха рекламной кампании в поисковых системах:

- выбор ключевых слов;
- ценовая политика;
- содержание рекламы;
- расположение рекламы;
- популярность страниц, размещающих объявление.

Интернет-каталоги (web directories) управляются редакторами и часто называются ручными («человеческими») поисковыми системами.

Результаты в каталогах дополняются «приходящими» (fallthrough) от партнерской поисковой системы.

Участие сайта в серьезных специализированных каталогах обеспечит лидирующие позиции в автоматических поисковых системах.

Специализированные поисковые системы: локальные, новостные, поиска товаров и услуг.

Электронный маркетинг:

- поисковая оптимизация;
- реклама в поисковых системах;
- платное размещение в поисковых системах;
- платное размещение в интернет-каталогах;
- оптимизация под специализированные (вертикальные) поисковые системы;
- оптимизация для мультимедийных поисковых систем.

Компоненты оптимальной и долгосрочной видимости сайта в поисковой системе: ключевые слова, ссылки, популярность. Эти факторы сильнее влияют на видимость в поисковых системах, чем использование ключевых слов в имени домена и/или в именах файлов.

Лучше всего поместить ключевые слова во все возможные HTML-теги, но избегая их переполнения: самые важные места сайта – теги title, тело сайта (видимый текст); текстовые гиперссылки, метатеги, альтернативный текст графических изображений, имена домена и файлов.

Проблематичные навигационные решения:

1. Небрежный HTML-код.
2. Графические меню навигации.
3. Ссылки, встроенные в скрипты JavaScript, динамические изображения, массивы, меню навигации со скриптами.
4. Динамические или основанные на базах данных веб-страницы с ?, &, \$, =, +, % в URL и генерируемые скриптами.
5. Flash.

Схема навигации должна позволять паукам обрабатывать ее содержимое: допустимо наличие двух схем на странице, одна из которых ориентирована на пользователей, а другая соответствует требованиям поисковой системы.

Дружественный поисковым системам сайт – это сайт, прежде всего дружественный пользователю, то есть тот, который легко найти с помощью автоматических, управляемых человеком и вертикальных поисковых систем.

Ссылки служат не только для навигации, но должны обеспечивать следующие параметры: доступность, релевантность, интуитивную ориентацию в пространстве сайта, интуитивную ориентацию в информационном поле сайта.

Поисковая оптимизация и юзабилити веб-сайтов не противоречат друг другу.

Компонент «Популярность»:

– популярность, или индекс цитирования ссылки в поисковых системах (link popularity);

– посещаемость ссылки, то есть частота переходов по щелчку на ссылке (click popularity, click-through popularity).

Качество ссылок с других сайтов на ваш сайт имеет гораздо больший «вес», нежели количество.

Крупнейшие поисковые системы и некоторые интернет-каталоги измеряют частоту нажатий пользователями ссылок на ваш сайт и время пребывания на нем (то есть чтения ваших страниц), а также частоту повторных посещений вашего сайта.

На данный момент уровень посещаемости ссылки не участвует в определении релевантности из-за clickbot programs.

Главная страница сайта обычно считается более важной, чем остальные.

Ссылок на главную страницу должно быть больше всего на других страницах.

Юзабилити влияет на SEO.

Базовые правила веб-дизайна:

- простота чтения;
- простота навигации;
- простота поиска;
- согласованность в макете и дизайне;
- высокая скорость загрузки.

В первую очередь глазом распознается желтый.

Уровень контрастности должен быть 90% и более. Более всего контрастируют черный и белый.

На графике – шрифты без засечек.

«Простота навигации» означает, что ваш целевой пользователь всегда должен знать, в какой части сайта он находится. На случай, если он заблудится, следует сделать карту сайта, справку, раздел поиска по сайту и кнопку возврата на главную страницу из любого места ресурса, чтобы пользователь мог определить, где находится сейчас, куда хотел бы перейти, где был до этого.

У пользователей должна быть возможность запросить поисковые системы и перейти на страницы именно с той информацией, которая им нужна.

Нужно обозначать цветом посещенные ссылки.

Цвета и эффекты гиперссылок должны быть всегда уникальными: другой текст не должен выглядеть так же.

На странице ЧаВо должны быть перечислены все вопросы сверху.

Должен быть беспрепятственный доступ к контактной информации.

Некоторые ключи имеют выраженную сезонность.

Мозговой штурм поможет выявить ключи, интересующие потенциальных клиентов, а не ваше начальство.

При вводе слов в строке запроса пользователя, как правило, не пользуются прописными буквами.

Не используйте рекламные и маркетинговые ключи.

В списке ключевых слов должно быть максимальное количество всевозможных комбинаций.

Обычно стоит уделить больше внимания длинным ключевым словосочетаниям, чем отдельным словам, поскольку те, кто делает специализированный запрос, являются более заинтересованными потенциальными клиентами.

Слова нужно прорешивать; задайтесь вопросами:

– в какой форме – в единственном или множественном числе – слово вводится чаще всего?

– какие 3–4 слова чаще всего вводят одновременно?

– в каком порядке вводят слова?

Многие поисковые системы или интернет-каталоги предоставляют связанные запросы поиска (related searches), похожие запросы (other searched for), narrow your search (сузить область поиска).

Инструменты: WebTrends, Omniture, ClickTracks, SearchMarketing (Yahoo), AdWords (Google), adCenter (Microsoft).

Количество поисковых запросов по какому-то конкретному ключу менее важно, чем получение максимума разнообразных вариантов его употребления.

Еще инструменты: Trellian, Wordtracker.

В FAQ – от 200 до 800 слов.

В title – 5–10 слов, или 60–90 знаков (без фильтруемых предлогов, союзов, частиц и так далее).

Ни в коем случае не помещайте бренд (название фирмы) в title (особенно в начало).

Аббревиатуры типа Inc., Ltd., ОАО, ЗАО, ООО никогда не задают в поиск.

Лучший способ придумать заголовок – следовать стратегии суммирования мощности power combo: при любой возможности помещайте на первые три позиции тега title такие слова, которые при вводе в строке поиска в любой комбинации составляют ключевое словосочетание – organic green teas = organic green teas, organic teas, green teas.

Поисковые системы оценивают также степень близости ключевых слов в title и всем первичном тексте сайта.

Пользователи ищут и множественное, и единственное число слов, поэтому в заголовках title можно их совмещать.

Все поисковые механизмы учитывают слова из верхней части веб-страницы, называемой обычно «верхней половиной полосы», которая для них важнее других участков страницы.

«Пауки» всегда будут индексировать для определения релевантности как теги title, так и теги body.

Высота расположения ключевых слов на странице называется заметностью ключевых слов.

Один из простых способов разместить ключевые слова в видимой области веб-страниц – включить их в заголовки.

Первый абзац вашей страницы должен точно описывать содержание всей этой страницы с применением наиболее желательных ключевых слов.

Эффективный способ оптимизации слишком длинных страниц – разместить список всех заголовков, содержащих ключевые слова в верхней части страницы – в видимой области. Другой способ заострить внимание на ключевых словах – писать заключительные абзацы или

предложения, размещаемые почти на каждой странице сайта, например, призывы к действию – calls to action.

Многие поисковые системы воспринимают текст гиперссылки и вокруг нее как важный.

Посетители сайта автоматически делают вывод, что синее слово с подчеркиванием (или просто слово с подчеркиванием) направит их прямо к информации, содержащейся в данном слове.

Ключевые слова лучше включать в гиперссылку.

Способ напоминания себе о том, что нужно встраивать в текст больше ключей, – вопросы типа «Какие ____?», «Что ____?», «О чем ____?» и так далее.

Только некоторые из крупнейших поисковых систем используют содержимое метатегов для определения релевантности.

Используйте в метатегах описания 4–5 наиболее важных ключевых слов на страницу и в общей сложности 200–250 символов.

Не забудьте выбросить из метатегов описания как можно больше фильтруемых элементов.

Если в метатегах описания вы используете слова, которые не встречаются на ваших веб-страницах, большинство поисковых систем сочтут их спамом.

Рекомендации для создания метатегов описания:

- не повторяйте в них в точности тот же текст, который использовался в содержимом тегов title. Это будет сигналом для поисковых роботов о возможном применении «набивки» содержимого ключами (keyword stacking);

- помещайте наиболее важное ключевое словосочетание в начало метатега описания;

- для важнейших ключевых слов по возможности используйте и единственное, и множественное число;

- старайтесь не разделять ключевые слова и/или словосочетания;

- по минимуму прибегайте к повторам ключевых слов. Не превышайте 3-4-х повторов;

- знайте, что некоторые поисковые системы воспринимают разные формы слова, например, единственного и множественного числа, как одно и то же слово;

- разделяйте повторяющиеся ключевые слова значимыми эпитетами;

- встраивайте призыв к действию, побуждая целевого пользователя щелкнуть на ссылке, ведущей на ваш сайт, чтобы «узнать подробнее о ...», где «...» – ключевые слова;

– перечень ключей, размещенный в метатеге описания, не только не заставит целевого пользователя нажать ссылку на ваш сайт, но также может послужить признаком потенциального спама для поисковых роботов;

– убедитесь, что ваши метатеги в основном содержат связные предложения или словосочетания. Если вы не поместили текст в метатеги описания, то поисковые системы, которые его используют, сами создают описание на основе содержимого страницы. Такое описание, возможно, отразит ваши страницы не в лучшем свете.

В метатеге ключевых слов keywords лучше всего помещать ключевые слова, содержащиеся в контенте страниц, иначе это послужит сигналом о том, что вы недобросовестно формируете контент для поисковых систем и пользователей. На что обратить внимание при выборе ключевых слов для атрибута метатега keywords:

– ключевые слова в вашем списке должны иметь ту форму числа, в которой их используют ваши посетители; более популярную форму слова располагайте в верхней части списка;

– большинство поисковых систем не различают размер букв;

– поисковые системы одинаково воспринимают в этом теге в качестве разделителя ключевых слов запятые и пробелы;

– вставлять слова без ошибок в орфографии, так как в теле сайта их нет.

Метатег revisit-after задает поисковому роботу инструкцию снова посетить данную страницу через определенный период. Но для повышения видимости это бесполезно.

Чтобы роботы не сканировали отдельные страницы, иногда используют метатег robots. Не все поисковые системы придают значение таким метатегам.

Альтернативный текст alt в теге img индексируется многими поисковыми системами.

Графические изображения в верхней части страницы более значимы, нежели изображения в нижней части страницы.

Альтернативный текст графического изображения должен быть релевантным реальному изображению и содержимому страницы, где оно размещено.

Плотность ключевых слов (keyword density) – мера определения количества ключевых слов на единицу текста: плотность ключей = количество ключей на странице / общее количество слов на странице.

Страница, содержащая от 250 до 800 слов (включая фильтруемые слова, но не включая метатеги или альтернативный текст), имеет оптимальное наполнение.

Некоторые поисковые системы не учитывают плотность ключевых слов на странице.

Подсвечивание ключевых слов в результатах поиска – highlighting.

Многие поисковики считают текст гиперссылки (anchor text) релевантным, им нравится возможность записать текст и внутри, и вокруг ссылок.

Специалисты по юзабилити рекомендуют в качестве вспомогательной схемы навигации поместить на страницу иерархическое дерево ссылок (breadcrumbs – «хлебные крошки») – это имеющаяся на каждой веб-странице строка текстовых гиперссылок, расположенных в порядке посещения пользователем страниц.

Активный текст – clickable.

Пользователей чаще привлекают шрифты, которых нет в установках по умолчанию.

Поисковики обычно не переходят по сенсорным изображениям (image map).

Главное преимущество выпадающих меню (drop-down menu) – экономия пространства на странице. Специалисты по юзабилити не рекомендуют использовать выпадающее меню.

Перекрестная ссылка (cross-linking) действует внутри сайта, связывая его страницу с другой его же страницей.

Продвижение ссылок (link development) связывает один сайт с совершенно другим сайтом.

Должны быть ясные ориентационные подсказки типа you are there («вы сейчас здесь») с ключами.

Люди склонны фокусировать взгляд ближе к центральной части экрана – на расстоянии 3–4 дюймов от верхней его границы. Там обычно дизайнеры размещают заголовки статей, и там же размещаются «хлебные крошки». В идеале каждый запрос пользователя к поисковой системе должен направлять его точно к тому разделу сайта, где содержится нужная ему информация.

? – знак динамического URL. В динамическом URL желательно минимизировать число параметров.

Лучше применять абсолютные (absolute) ссылки вместо относительных (relative) на случай возникновения поддоменов.

Поисковые роботы автоматически конвертируют относительные ссылки в абсолютные.

Поисковые системы не используют содержимое атрибута title гиперссылки для определения релевантности, но оно поможет улучшить интуитивную информационную ориентацию в тех браузерах, где текст атрибута title отображается при наведении указателя мыши на ссылку.

Проблемные символы URL?, &, \$, =, +, % можно заменить»,» и “/».

Дорвеи (doorway) – одна из форм спама. А информационные страницы приемлемы для поисковиков и интернет-каталогов. Тексты дорвеев предназначены для достижения лидирующих позиций и генерируются компьютерами. Информационные страницы, напротив, целенаправленно стремятся удовлетворить посетителей сайта, поскольку содержат полезную и интересную информацию. Дорвеи применяют redirect, поэтому нуждаются в клоакинге (cloaking).

searchenginewatch.com.

Поисковики не включают страницы с идентификатором сеанса в URL в индексы.

robots.txt – протокол запрета сканирования (robots exclusion protocol). Протокол запрета сканирования нужно размещать на сервере до размещения запрещенного им контента, чтобы избежать появления ненужной информации в результатах поиска.

Популярность ссылки, или индекс цитирования ссылки (link popularity) на веб-страницу, определяется количеством и качеством ссылок на данную страницу, размещенных на внешних ресурсах.

Крупнейшие поисковые системы уже отказались от измерения количества ссылок, которые имеет какой-либо сайт.

«Фермы ссылок» – free-for-all link farms.

Не полагайтесь на мнение отдела маркетинга или команды дизайнеров в отношении предпочтений целевых пользователей без специального тестирования эффективности ваших веб-страниц.

Звуковые эффекты и Flash могут мешать поисковой оптимизации. Тег noscript обеспечивает альтернативное содержимое для браузеров, не поддерживающих JavaScript, и пользователей, запретивших JavaScript в своих браузерах. Тег noscript следует размещать внутри тега head. Никогда не используйте тег noscript для скрытия какого-либо текста, не относящегося к содержимому страницы, или ссылок, которые вы не хотите показывать своим пользователям.

Нельзя дублировать тег title.

Страницы с фреймами снабжайте тегом noframes.

Поисковики снижают релевантность текста внутри noscript и noframes.

Должно быть минимум две схемы навигации: для пользователей и для поисковиков.

С отказом от фреймов поток посетителей, приходящих из поисковых систем, обычно возрастает.

Коэффициент рентабельности – return on investment, ROI.

Для подозрительных ссылок задавайте атрибут nofollow.

Формат GIF поддерживается практически всеми браузерами.
Около 15–20% всех запросов в поисковых системах имеют отношение к графическим изображениям.
Для графики важно название файла.

Полезные высказывания из книги «Совершенный код» Макконнелла

В разделе приведены цитаты из [12].

Средство распространения информации о хороших методиках сыграло бы важную роль.

Для широкого распространения исследовательских разработок требуется от 5 до 15 и более лет.

Ежегодно программистами становятся около 50000 человек.

Число дипломов, вручаемых в отрасли, около 35000.

Единственным способом получения информации является изучение горы книг и нескольких сотен технических журналов, дополненное значительным реальным опытом (о конструировании ПО).

Сайт книги cc2e.com/1234.

Конструирование ПО = кодирование.

На конструирование приходится 65% в больших проектах.

Во время конструирования допускается от 50 до 75% ошибок в больших проектах.

Ошибки конструирования дешевле исправлять.

Одними из самых дорогих ошибок в истории, приведшими к убыткам в сотни миллионов долларов, были мелкие ошибки кодирования.

При написании программы только этапа конструирования нельзя избежать.

Почта и сайт автора: stevencc@construx.com, stevemccconnell.com.

Книги никогда не создаются в одиночку.

Составляющие разработки ПО за последние 25 лет:

- определение проблемы;
- выработка требований;
- создание плана конструирования;
- разработка архитектуры ПО, или высокоуровневое проектирование;
- детальное проектирование;
- кодирование и отладка;
- блочное тестирование;
- интеграционное тестирование;
- интеграция;
- тестирование системы;
- корректирующее сопровождение.

Проекты бывают формальные, неформальные.

Конструирование не включает определение проблемы.

Конструирование = программирование.

В центре конструирования – кодирование и отладка.

Конкретные задачи, связанные с конструированием:

1. Проверка выполнения условий, необходимых для успешного конструирования.

2. Определение способов последующего тестирования кода.

3. Проектирование и написание классов и методов.

4. Создание и присвоение имен переменным и именованным константам.

5. Выбор управляющих структур и организация блоков команд.

6. Блочное тестирование, интеграционное тестирование и отладка собственного кода.

7. Взаимный обзор кода и низкоуровневых программных структур членами группы.

8. «Шлифовка» кода путем его тщательного форматирования и комментирования.

9. Интеграция программных компонентов, созданных по отдельности.

10. Оптимизация кода, направленная на повышение его быстродействия, и снижение степени использования ресурсов.

В конструирование не входят управление, выработка требований, разработка архитектуры, проектирование пользовательского интерфейса, тестирование системы и ее сопровождение.

На конструирование обычно уходит 30–80% времени работы.

Требования к приложению и архитектура разрабатываются, чтобы гарантировать эффективность этапа конструирования.

Тестирование системы выполняется после конструирования, чтобы проверить его правильность.

Производительность труда отдельных программистов во время конструирования изменяется в 10–20 раз.

Исходный код – самая свежая документация на ПО.

Тестирование системы должно контролироваться статистически.

Повышение эффективности конструирования позволяет оптимизировать любой проект.

Компетентность в конструировании ПО определяет то, насколько хороший вы программист.

Метафоры позволяют лучше понять разработку ПО.

Использование метафор (аналогий) – моделирование.

Эффективность моделей объясняется их яркостью и концептуальной целостностью.

При чрезмерном обобщении модели вводят в заблуждение.

Хорошие метафоры простые, согласуются с другими метафорами и объясняют многие данные и явления.

Описываемое метафорами поведение предсказуемо и понятно всем людям.

Самая сложная часть программирования – концептуализация проблемы.

90% общего объема работы над программной системой выполняется после выпуска первой версии.

Популярные метафоры о разработке ПО:

1. Написание письма.
2. Выращивание сельскохозяйственных культур.
3. Процесс формирования жемчужины.
4. Построение дома (невыгодно писать компоненты, которые можно купить готовыми).

Программная система из 1 млн строк требует 69 видов документации, спецификация требований занимает 4000– 5000 страниц.

В неудачных проектах конструирование приходится выполнять несколько раз.

Вайнберг «The psychology of computer programming».

Хороших программистов всегда не хватает.

Последовательный подход (вопросы решаются заблаговременно) применяется, когда:

- требования довольно стабильны;
- проект приложения прост и относительно понятен;
- группа разработчиков знакома с прикладной областью;
- проект не связан с особым риском;
- важна долговременная предсказуемость проекта;
- затраты на изменение требований, проекта приложения и кода, скорее всего, окажутся высокими.

Итеративный подход (вопросы решаются по мере работы) применяется, когда:

- требования относительно непонятны или вам кажется, что они могут оказаться нестабильными по другим причинам;
- проект приложения сложен, не совсем ясен или и то и другое;
- группа разработчиков незнакома с прикладной областью;
- проект сопряжен с высоким риском;
- долговременная предсказуемость проекта не играет особой роли;
- затраты на изменение требований, проекта приложения и кода, скорее всего, будут низкими.

Итеративные подходы эффективны чаще.

Предварительное условие: ясное формулирование проблемы, которую система должна решать (без намека на решение).

Процесс программирования:

- 1) определение проблемы;
- 2) выработка требований;
- 3) проектирование архитектуры;
- 4) конструирование;
- 5) тестирование системы;
- 6) будущие улучшения.

Проблема должна быть описана с пользовательской точки зрения (без компьютерных терминов).

Требования подробно описывают, что должна делать программная система.

Функциональность системы определяется пользователем, а не программистом (требования решают споры, сводят к минимуму изменения системы после начала разработки, снижают дополнительные расходы).

Процесс разработки помогает разработчикам и клиентам лучше понять свои потребности.

В среднем проекте требования во время разработки изменяются на 25%, на что приходится 80% повторной работы над проектом.

Изменение требований влечет пересмотр графика и сметы.

evolutionary prototyping – поставка системы клиенту по частям.

Определить моменты прекращения работы над проектом.

Архитектура описывает, какие другие компоненты данный компонент может использовать непосредственно, какие косвенно, а какие вообще не должен использовать; должна описывать организацию классов в подсистеме и обосновывать итоговый вариант.

Интерфейс проектируется на этапе выработки требований либо в архитектуре.

Архитектура должна быть модульной.

Архитектура определяет подходы к безопасности.

В требованиях определяются показатели производительности.

Масштабируемость – возможность системы адаптироваться к росту требований.

Интернационализация – реализация в программе поддержки региональных стандартов.

Локализация – перевод интерфейса программы и реализация в ней поддержки конкретного языка.

Архитектура выражает отношение к избыточной функциональности.

Архитектура четко описывает стратегию изменений.

В архитектуре должны быть обоснованы важнейшие принятые решения.

Цели должны быть четко сформулированы.

В архитектуре явно определены области риска, указаны его причины и описаны действия по сведению риска к минимуму.

В проекте без проблем работа над требованиями, архитектурой и предварительным планированием поглощает 20–30% времени.

Программисты, использующие язык, с которым они работали три года или более, примерно на 30% более продуктивны, чем программисты, обладающие аналогичным опытом, но для которых язык является новым.

Программисты, использующие языки высокого уровня, достигают более высокой производительности и создают более качественный код, чем программисты, работающие с языками низкого уровня.

Гипотеза Сапира-Уорфа: способность человека к обдумыванию определенных мыслей зависит от знания слов, при помощи которых можно выразить эту мысль.

Ada, Cobol используются в Минобороны США.

SQL – декларативный язык.

Программируют с использованием языка, а не на языке.

Важный аспект работы проектировщика – анализ конкурирующих характеристик проекта и достижение баланса между ними.

Спроектировать программу можно десятками разных способов.

Существенные свойства – которыми объект должен обладать, чтобы быть именно этим объектом.

Несущественные (акцидентные) свойства – которые не влияют на суть объекта.

Управление сложностью – самый важный аспект разработки ПО.

Э. Дейкстра: «Ни один человек не обладает интеллектом, способным вместить все детали современной компьютерной программы. Поэтому целью является минимизация объема программы, о котором нужно думать в конкретный момент времени».

Цель всех методик проектирования ПО – разбиение сложной проблемы на простые фрагменты.

Краткость методов помогает снизить нагрузку на интеллект. Этому же способствует написание программы в терминах проблемной области, а также работа на самом высоком уровне абстракции.

Как бороться со сложностью? Чаще всего причинами неэффективности являются:

- сложное решение простой проблемы;
- простое, но неверное решение сложной проблемы;

- неадекватное сложное решение сложной проблемы.

Дейкстра: «Сложность современного ПО обусловлена самой его природой».

Двойственный подход к управлению сложностью:

- старайтесь свести к минимуму объем существенной сложности, с которым придется работать в каждый конкретный момент времени;
- сдерживайте необязательный рост несущественной сложности.

Внутренние характеристики проекта:

- минимальная сложность;
- простота сопровождения;
- слабое сопряжение;
- расширяемость;
- возможность повторного использования;
- система предусматривает интенсивное использование вспомогательных низкоуровневых классов;
- класс использует минимальное число других классов;
- портируемость;
- минимальная, но полная функциональность (дополнительный код необходимо разработать, проанализировать, протестировать, пересматривать при изменении других фрагментов программы, поддерживать совместимость будущих версий с дополнительным кодом);
- систему можно изучать на отдельных уровнях, игнорируя другие уровни (проектировать дополнительный уровень, скрывающий плохое качество старого кода);
- соответствие стандартным популярным подходам.

Уровни детальности проектирования программной системы: – вся система;

- разделение системы на подсистемы или пакеты, определение взаимодействий между ними (минимальные взаимодействия для облегчения модификации (проще сначала ограничить взаимодействие, а затем сделать его более свободным)).

Отношения между системами должны быть простыми. В порядке уменьшения простоты:

- 1) одна подсистема вызывает методы другой;
- 2) одна подсистема содержит классы другой;
- 3) наследование классов одной подсистемы от классов другой.

Программа не должна содержать циклических отношений.

Часто используемые подсистемы: – бизнес-правил;

- пользовательского интерфейса;
- доступа к БД;
- изоляции зависимостей от ОС.

Разделение подсистем на классы обычно требуется во всех проектах.

Объект – динамическая сущность, класс – статическая.

БД: различие между классом и объектом аналогично различию между «схемой» и «экземпляром».

Разделение классов на методы необходимо в каждом проекте и часто оставляется отдельным программистам.

Проектирование методов может включать написание псевдокода, поиск алгоритмов в книгах, размышление над оптимальной организацией фрагментов метода и написание кода.

При проектировании с использованием искусственных объектов и объектов реального мира определите:

- объекты и их атрибуты (методы и данные);
- действия, которые могут быть выполнены над каждым объектом;
- действия, которые каждый объект может выполнять над другими объектами (включение и наследование);
- части каждого объекта, видимые другим объектам, то есть открытые и закрытые части;
- открытый интерфейс каждого объекта (на уровне языка программирования).

Открытый интерфейс – данные и методы, которые объект предоставляет в распоряжение остальным объектам.

Защищенный интерфейс – части объекта, доступные производным от него объектам.

2 вида итерации:

- высокоуровневая, направленная на улучшение организации классов;
- на уровне определенных классов, направленная на детализацию проекта каждого класса.

Дом – абстракция его составляющих.

Абстракция – один из главных способов борьбы со сложностью реального мира.

Абстракция позволяет задействовать концепцию, игнорируя ее некоторые детали и работая с разными деталями на разных уровнях.

Создавать абстракции на уровне интерфейсов методов, интерфейсов классов и интерфейсов пакетов.

Инкапсуляция не только представляет сложную концепцию в более простой форме, но и не позволяет взглянуть на какие бы то ни было детали сложной концепции.

Наследование позволяет создать универсальные методы для выполнения всего, что основано на общих свойствах дверей, и затем написать

специфические методы для выполнения специфических операций над конкретными типами дверей.

Поддержка языком операций вроде Open () или Close () при отсутствии информации о конкретном типе двери вплоть до периода выполнения называется полиморфизмом.

Интерфейсы классов должны быть полными и минимальными, должны сообщать как можно меньше о внутренней работе

класса (класс похож на айсберг, большая часть которого скрыта под водой).

Разработка интерфейса класса – итеративный процесс (пытаться создать или использовать другой подход).

Соккрытие аспектов проектирования позволяет значительно уменьшить объем кода, затрагиваемого изменениями (например, применение именованных констант вместо литералов; соккрытие информации: «id=new Id ();» вместо «id=++g_maxId;»).

Категории секретов:

- секреты, которые скрывают сложность, позволяя программистам забыть о ней при работе над остальными частями программы;

- секреты, которые скрывают источники изменений с целью локализации результатов возможных изменений.

Барьеры, препятствующие соккрытию информации:

- избыточное распространение информации (например, использование литералов вместо констант, распространение кода взаимодействия с пользователем по системе – GUI концентрировать в одном классе (пакете, подсистеме));

- круговая зависимость (В зависит от А, который зависит от В; не позволяет протестировать один класс, пока не будет готова часть другого);

- ошибочное представление о данных класса как о глобальных данных;

- кажущееся снижение производительности.

Крупные программы, использующие соккрытие информации, в 4 раза легче модифицировать, чем программы, его не использующие.

Размышлять о том, что скрыть.

Подготовка к изменениям:

1. Определите элементы, изменение которых кажется вероятным.

2. Изолируйте элементы, изменение которых кажется вероятным.

Интерфейс класса должен скрывать изменения в классе.

Области, изменяющиеся чаще всего:

- бизнес-правила;

- зависимости от оборудования;

- ввод-вывод;
- нестандартные возможности языка;
- сложные аспекты проектирования и конструирования;
- переменные статуса.

В качестве переменных статуса применять не булевы переменные, а перечисления.

Вместо непосредственной проверки переменной использовать методы доступа.

Проектировать систему так, чтобы влияние изменений было обратно пропорционально их вероятности.

Если изменение маловероятно, но его легко предугадать, рассмотрите его внимательнее, чем более вероятное изменение, которое трудно спланировать.

Функции, нужные пользователям, – ядро системы, которое не требуется изменять.

Путем небольших приращений экстраполировать систему, определяя дополнительную часть.

Сопряжение характеризует силу связи класса или метода с другими классами или методами.

Поддерживать сопряжение слабым.

Слабое сопряжение (*loose coupling*) – классы и методы имеют немногочисленные, непосредственные, явные и гибкие отношения с другими классами.

Эффективный механизм соединения максимально прост.

Отношения модулей должны напоминать отношения деловых партнеров, а не сямских близнецов.

Критерии оценки сопряжения:

- объем связи (число соединений между модулями – число параметров метода, число открытых методов);
- видимость (заметность связи между двумя модулями);
- гибкость (легкость изменения связи между модулями).

Самые распространенные виды сопряжения:

- простое сопряжение посредством данных-параметров (передача элементарных типов данных через списки параметров);
- простое сопряжение посредством объекта (модуль создает экземпляр объекта, с которым сопряжен);
- сопряжение посредством объекта-параметра (объект 1 требует, чтобы объект 2 передал ему объект);
- семантическое сопряжение (один модуль использует семантические знания о внутренней работе другого модуля): модуль 1 передает в модуль 2 управляющий флаг, определяющий дальнейшую работу моду-

ля 2; модуль 2 использует глобальные данные после их изменения модулем 1; семантическое сопряжение опасно тем, что изменение кода в используемом модуле может так нарушить работу использующего модуля, что компилятор этого не определит.

Классы и методы должны упрощать работу.

Шаблоны снижают сложность, предоставляя готовые абстракции, снижают число ошибок, стандартизируя детали популярных решений.

Шаблоны имеют эвристическую ценность, указывая на возможные варианты проектирования.

Шаблоны упрощают взаимодействие между разработчиками, позволяя им общаться на более высоком уровне.

Ловушки шаблонов – насильственная адаптация кода к шаблону, применение шаблона, продиктованное не целесообразностью, а желанием испытать шаблон в деле.

Шаблоны проектирования – эффективный инструмент управления сложностью.

Шаблон – это эвристический принцип.

Связность (cohesion) должна быть максимальна.

Связность характеризует то, насколько хорошо все методы класса или все фрагменты метода соответствуют главной цели.

Формируйте иерархии.

Иерархия – это многоуровневая структура организации информации, при которой наиболее общая или абстрактная репрезентация концепции соответствует вершине, а более детальные специализированные репрезентации – более низким уровням.

Люди в целом находят иерархии естественным способом организации сложной информации.

Рисуя сложный объект, люди рисуют его иерархически.

Формализовать контракты классов (обещания клиентов – предусловия, обязательства класса – постусловия).

Грамотно назначать сферы ответственности.

Проектировать систему для тестирования.

Тщательно рассматривать возможные причины аварий, а не просто копировать другие успешные проекты.

Тщательно выбирать время присвоения переменной конкретного значения (binding time).

Создавать центральные точки управления – управление может быть централизовано в классах, методах, макросах препроцессора, файлах библиотек.

Рисовать диаграммы – они представляют проблему на более высоком уровне абстракции.

Поддерживать модульность проекта системы – каждый метод или класс должен быть похож на черный ящик (известны входы и выходы, но неизвестно, что внутри).

Проектировать ПО нужно с использованием разных подходов.

Проектирование – итеративный процесс. После его завершения нужно возвращаться к началу.

Лучше обнаруживать варианты, которые не работают, чем ничего не делать.

Нисходящее (top-down) проектирование начинается на высоком уровне абстракции.

Восходящее (bottom-up) начинается со специфики и постепенно переходит ко все большей общности.

Если сейчас решение кажется вам чуть-чуть хитрым, для любого, кто будет работать над ним позднее, оно станет головоломкой.

Нисходящая стратегия – декомпозиция. Восходящая – композиция.

Большинство людей находят разбиение крупной концепции на меньшие части более легким, чем объединение небольших концепций в более крупную.

Прототипирование работает плохо, если задача недостаточно конкретна.

Имена прототипных классов и пакетов должны иметь префикс `prototype`.

Механические действия вытесняют творчество.

Регистрировать протоколы обсуждения проекта и принятые решения при помощи Wiki.

Отправлять резюме дискуссий всем членам группы по электронной почте.

Фотографировать схемы на доске.

Хранить плакаты со схемами проекта.

Использовать UML.

Класс – это набор данных и методов, имеющих общую, целостную, хорошо определенную сферу ответственности.

Классы нужны для максимизации части программы, которую можно игнорировать при работе над конкретными фрагментами кода.

Абстрактный тип данных – это набор, включающий данные и выполняемые над ними операции (служащие одной цели).

Преимущества использования АД:

- возможность сокрытия деталей реализации;
- ограничение области изменений;
- более высокая информативность интерфейса;
- легкость оптимизации кода;

- легкость проверки кода;
- удобочитаемость и понятность кода;
- ограничение области использования данных;
- возможность работы с сущностями реального мира, а не с низкоуровневыми деталями реализации.

Принципы использования АД:

- представлять в форме АД распространенные низкоуровневые типы данных;
- представлять в форме АД часто используемые объекты, такие как файлы;
- представлять в форме АД даже простые элементы;
- обращайтесь к АД так, чтобы это не зависело от среды, используемой для его хранения.

В ООП каждый АД можно реализовать как класс (класс дополнительно поддерживает наследование и полиморфизм).

Интерфейс класса – это абстракция реализации класса, скрытой за интерфейсом.

Принципы проектирования интерфейсов:

- выражать в интерфейсе класса согласованный уровень абстракции (смешанные уровни абстракции делают программу все менее и менее понятной);
- убедиться в понимании того, реализацией какой абстракции является класс;
- представляйте методы вместе с противоположными им методами (создавать противоположные методы, не имея на то причин, не следует);
- убирать постороннюю информацию в другие классы;
- преобразовывать семантические элементы интерфейса в программные, например, утверждениями (интерфейсы должны как можно меньше зависеть от документации);
- элементы интерфейса должны находиться на одном уровне абстракции;
- не включать в класс открытые члены, плохо согласующиеся с абстракцией интерфейса;
- рассматривать абстракцию и связность вместе.

Хорошая инкапсуляция:

- минимизировать доступность классов и их членов;
- не делать данные-члены открытыми;
- не включать в интерфейс класса закрытые детали реализации;
- класс не должен делать предположений о том, как этот интерфейс будет или не будет использоваться;

- избегать использования дружественных классов;
- не делать метод открытым лишь потому, что он использует только открытые методы;
- ценить легкость чтения кода выше, чем удобство его написания;
- избегать зависимости клиентского кода от закрытой реализации класса, а не от его открытого интерфейса (должен быть интерфейс, позволяющий разобраться, не глядя на реализацию);
- остерегаться жесткого сопряжения – сильной связи между двумя классами.

Включение (containment) – один класс содержит примитивный элемент данных или другой класс.

Включение проще наследования и меньше подвержено ошибкам.

Включение можно реализовать отношением «содержит» (в крайнем случае – закрытое наследование).

Класс должен содержать 9 примитивных типов или 5 сложных объектов.

Наследование помогает избежать повторения кода и данных в нескольких местах, централизуя их в базовом классе (один класс является более специализированным вариантом другого класса).

Проектировать и документировать классы с учетом возможности наследования или запретить его.

Наследование повышает сложность программы.

Все методы базового класса должны иметь в каждом производном классе то же значение.

Убедиться, что наследуется только то, что нужно.

Не используйте имена непереопределяемых методов базового класса в производных классах.

Перемещайте общие интерфейсы, данные и формы поведения на как можно более высокий уровень иерархии наследования, если это не нарушит абстракцию.

С подозрением относитесь к классам, объекты которых создаются в единственном экземпляре (возможно, класс перепутан с объектом).

С подозрением относитесь к базовым классам, имеющим только один производный класс.

С подозрением относитесь к классам, которые переопределяют метод, оставляя его пустым.

Избегайте многоуровневых иерархий наследования, ограничивая иерархии наследования максимум 6 уровнями.

Предпочитайте полиморфизм, а не крупномасштабную проверку типов.

Делайте все данные закрытыми, а не защищенными.

Миксин – простой класс, позволяющий добавлять ряд свойств в другой класс.

Ромбовидная схема наследования – классическая проблема.

Используйте множественное наследование, только тщательно рассмотрев все альтернативные варианты и проанализировав влияние выбранного подхода на сложность и понятность системы.

Ради управления сложностью относитесь к наследованию с подозрением.

Включайте в класс как можно меньше методов.

Блокируйте неявные методы и операторы, которые вам не нужны (private).

Минимизируйте число разных методов, вызываемых классом.

Избегайте опосредованных вызовов методов других классов (цепочек вызовов).

Вообще минимизируйте сотрудничество класса с другими классами.

Инициализируйте по мере возможности все данные-члены во всех конструкторах.

Создавайте классы-одиночки с помощью закрытого конструктора.

Если сомневаетесь, выполняйте полное копирование, а не ограниченное.

Разумные причины создания класса:

- моделирование объектов реального мира;
- моделирование абстрактных объектов;
- снижение сложности;
- изоляция сложности;
- сокрытие деталей реализации;
- ограничение влияния изменений;
- сокрытие глобальных данных;
- упрощение передачи параметров в метод;
- создание центральных точек управления;
- облегчение повторного использования кода;
- планирование создания семейства программ;
- упаковка родственных операций;
- выполнение специфического вида рефакторинга.

Классы, создавать которые не следует:

– избегайте создания классов, которые все знают и все могут;

– если класс имеет только данные, но не формы поведения, то это не класс;

– если класс имеет только формы поведения, но не данные, то это не класс.

Аспекты классов, зависящие от языка:

- поведение переопределенных конструкторов и деструкторов в дереве наследования;
- поведение конструкторов и деструкторов при обработке исключений;
- важность конструкторов по умолчанию (конструкторов без аргументов);
- время вызова деструктора или метода финализации;
- целесообразность переопределения встроенных операторов языка, в том числе операторов присваивания и сравнения;
- управление памятью при создании и уничтожении объектов или при их объявлении и выходе из области видимости.

В настоящее время использование классов – лучший способ достижения модульности.

Метод – это отдельная функция или процедура, выполняющая одну задачу.

Имя метода говорит о его роли.

Метод должен быть документирован, форматирован.

Входные переменные метода не изменяются.

Метод не работает с глобальными переменными.

Метод имеет одну четко определенную цель, должен быть защищен от получения плохих данных, не использует магические числа; все параметры используются в методе, параметров у метода не более 7, параметры метода упорядочены.

Методы – самый эффективный способ уменьшения объема и повышения быстродействия программ.

Разумные причины создания методов:

- снижение сложности;
- формирование понятной промежуточной абстракции;
- предотвращение дублирования кода;
- переопределить небольшой грамотно организованный метод легче, чем длинный и плохо спроектированный;
- сокрытие очередности действий;
- сокрытие неудобочитаемых операций;
- изоляция непортируемого кода;
- упрощение сложных булевых проверок;
- облегчение определения неэффективных фрагментов кода.

Один из главных ментальных барьеров, препятствующих созданию эффективных методов, – нежелание создавать простой метод для простой цели.

Высокая связность хуже низкой.

Функциональная связность – метод выполняет одну и только одну операцию.

Последовательная связность – метод содержит операции, которые обязательно выполняются в определенном порядке, используют данные предыдущих этапов и не формируют в целом единую функцию.

Коммуникационная связность – выполняемые в методе операции используют одни и те же данные и не связаны между собой иным образом.

Временная связность – когда операции объединяются в метод на том основании, что все они выполняются в один интервал времени.

Не выполнять в методе конкретные операции непосредственно, а вызывать для их выполнения другие методы.

Неприемлемые виды связности:

Процедурная связность – когда операции в методе выполняются в определенном порядке.

Поместить разные операции в разные методы.

Логическая связность – метод включает несколько операций, а выбор выполняемой операции осуществляется на основе передаваемого в метод управляющего флага.

Случайная связность – каких-либо ясных отношений между выполняемыми в методе операциями нет.

Стремиться создавать методы с функциональной связностью.

Советы по выбору удачных имен методов:

- описывайте все, что метод выполняет (методы с побочными эффектами избавлять от побочных эффектов);

- избегайте невыразительных и неоднозначных глаголов (роль метода должна быть очевидной);

- не используйте для дифференциации имен методов исключительно номера;

- не ограничивайте длину имен методов искусственными правилами (оптимальная длина имени переменной равняется в среднем 9–15 символам, но имена методов обычно длиннее из-за их сложности);

- для именования функции используйте описание возвращаемого значения;

- для именования процедуры используйте выразительный глагол, дополняя его объектом (кроме ООП языков);

- дисциплинированно используйте антонимы (add/remove, begin/end, create/destroy, first/last, get/put, get/set, increment/ decrement, insert/delete, lock/unlock, min/max, next/previous, old/new, open/close, show/hide, source/target, start/stop, up/ down);

– определяйте конвенции именования часто используемых операций.

Код требует минимальных изменений, если методы состоят в среднем из 100–150 строк.

Предотвращение ошибок коммуникации между методами:

- передавайте параметры в порядке «входные значения – изменяемые значения – выходные значения»;
- подумайте о создании собственных ключевых слов in и out;
- документируйте выраженные в интерфейсе предположения о параметрах.

Типы предположений:

- вид параметров: являются ли они исключительно входными, изменяемыми или исключительно выходными;
- единицы измерения (дюймы...);
- смыслы кодов статуса и ошибок, если для их представления не используются перечисления;
- диапазоны допустимых значений;
- специфические значения, которые никогда не должны передаваться в метод;
- ограничивайте число параметров метода примерно семью;
- подумайте об определении конвенции именования входных, изменяемых и выходных параметров;
- передавайте в метод те переменные или объекты, которые нужны ему для поддержания абстракции интерфейса;
- сопряжение методов должно быть минимальным;
- используйте именованные параметры;
- убедитесь, что фактические (переданные в метод) параметры соответствуют формальным (объявленные в методе);
- проверяйте все возможные пути возврата значения из функции;
- инициализируйте возвращаемое значение значением по умолчанию в начале функции;
- не возвращайте ссылки или указатели на локальные данные (вместо ссылок – данные-члены).

Макросы:

- разрабатывая макрос, заключайте в скобки все, что можно;
- заключайте макрос, включающий несколько команд, в фигурные скобки;
- не заменяйте методы макросами;
- называйте макросы, расширяющиеся в код подобно методам, так, чтобы при необходимости их можно было заменить методами;

– теоретически встраивание методов может повысить быстродействие;

– не злоупотребляйте встраиваемыми методами.

Защита от неправильных входных данных:

– проверяйте все данные из внешних источников;

– проверяйте значения всех входных параметров метода;

– решите, как обрабатывать неправильные входные данные.

Утверждение – это код, используемый во время разработки, с помощью которого программа проверяет правильность своего выполнения (логическое выражение + сообщение).

Положения по применению утверждений:

– используйте процедуры обработки ошибок для ожидаемых событий и утверждения для событий, которые происходить не должны;

– старайтесь не помещать выполняемый код в утверждения;

– используйте утверждения для документирования и проверки предусловий и постусловий;

Предусловия – это соглашения, которые клиентский код, вызывающий метод или класс, обещает выполнить до вызова метода или создания экземпляра объекта.

Постусловия – это соглашения, которые метод или класс обещает выполнить при завершении своей работы.

Для большей устойчивости кода проверяйте утверждения, а затем все равно обработайте возможные ошибки.

Способы обработки ошибок:

– вернуть нейтральное значение. В больших долгоживущих системах различные части могут разрабатываться несколькими проектировщиками 5–10 лет и более.

– заменить следующим корректным блоком данных;

– вернуть тот же результат, что и в предыдущий раз;

– подставить ближайшее допустимое значение;

– записать предупреждающее значение в файл;

– вернуть код ошибки;

– вызвать процедуру или объект-обработчик ошибок;

– показать сообщение об ошибке, где бы она ни случилась;

– обработать ошибку в месте возникновения наиболее подходящим способом;

– прекратить выполнение.

Корректность предполагает, что нельзя возвращать неточный результат; устойчивость требует всегда пытаться сделать что-то, что позволит программе продолжить работу.

Надо стараться реагировать на неправильные значения параметров одинаково во всей программе.

Предложения по исключениям:

- используйте исключения для оповещения других частей программы об ошибках, которые нельзя игнорировать;
- генерируйте исключения только для действительно исключительных ситуаций;
- не используйте исключения по мелочам (стараться обрабатывать ошибки локально);
- избегайте генерировать исключения в конструкторах и деструкторах, если только вы не перехватываете их позднее;
- генерируйте исключения на правильном уровне абстракции;
- вносите в описание исключения всю информацию о его причинах;
- избегайте пустых блоков `catch`;
- выясните, какие исключения генерирует используемая библиотека;
- рассмотрите вопрос о централизованном выводе информации об исключениях;
- стандартизируйте использование исключений в вашем проекте;
- рассмотрите альтернативы исключениям;
- преобразовывайте данные к нужному типу в момент ввода;
- методы с внешней стороны баррикады должны использовать обработчики ошибок, поскольку небезопасно делать любые предположения о данных; методы внутри баррикад должны использовать утверждения, так как данные, переданные им, считаются проверенными;
- промышленная версия: должна работать быстро, экономна с ресурсами, не позволяет пользователю делать опасные действия; отладочная версия: может работать медленно, может быть расточительной, может предоставлять дополнительные возможности без риска нарушить безопасность;
- внедрите поддержку отладки как можно раньше;
- исключительные случаи должны обрабатываться так, чтобы во время разработки они были очевидны, а в промышленном коде – позволяли продолжить работу;
- чем жестче требования во время разработки, тем проще эксплуатация;
- используйте встроенный препроцессор;
- используйте отладочные заглушки;
- оставьте код, который проверяет существенные ошибки;
- удалите код, проверяющий незначительные ошибки;
- удалите код, приводящий к прекращению работы программы;

– если на стадии разработки ваша программа обнаруживает ошибку, ее надо сделать незаметнее, чтобы ее могли исправить; -оставьте код, который позволяет аккуратно завершить работу программы;

– регламентируйте ошибки для отдела технической поддержки;

– убедитесь, что оставленные сообщения об ошибках дружелюбны (выводить телефон и адреса электронной почты, по которым можно о ней сообщить).

Обычно каждый метод тестируется при его создании.

Этапы конструирования классов:

1. Создание общей структуры класса.
2. Конструирование процедур класса.
3. Оценка и тестирование всего класса.

Действия по созданию метода:

1. Проектирование метода.
2. Проверка структуры.
3. Кодирование метода.
4. Пересмотр и тестирование кода.

Применение псевдокода:

– применяйте формулировки, в точности описывающие отдельные действия;

– избегайте синтаксических элементов языков программирования;

– описывайте назначение подхода, а не то, как этот подход нужно реализовать на выбранном языке программирования;

– пишите псевдокод на достаточно низком уровне, так, чтобы код из него генерировался практически автоматически.

Сформулируйте задачу, решаемую методом, настолько детально, чтобы можно было переходить к созданию метода.

Затруднения в выборе имени метода могут свидетельствовать о том, что его назначение не совсем понятно.

Метод должен иметь понятное, недвусмысленное имя.

В процессе написания метода думайте о том, как вы будете его тестировать.

Исследуйте функциональность, представляемую стандартными библиотеками.

Не тратьте время на реализацию готового алгоритма, по которому написана кандидатская диссертация.

Подумайте обо всем плохом, что может случиться с вашим методом.

Не теряйте времени на вылизывание отдельных методов, пока не выяснится, что это необходимо.

Если доступные библиотеки не предоставляют нужной функциональности, имеет смысл исследовать литературу с описанием алгоритмов.

Общая идея: раз за разом проходиться по псевдокоду, пока каждое его предложение не станет настолько простым, что под ним можно будет вставить строку программы, а псевдокод оставить в качестве документации.

Этапы кодирования метода:

1. Напишите объявление метода.
2. Напишите первый и последний операторы, а псевдокод превратите в комментарии верхнего уровня.
3. Добавьте код после каждого комментария.
4. Проверьте код.
5. Исправьте неточности.

Длина абзаца кода, как и длина абзаца литературного текста, зависит от высказываемой мысли, а его качество – от понимания автором сути.

Проверьте, не нужна ли дальнейшая декомпозиция кода.

Умозрительно проверьте ошибки в методе.

Только около 5% всех ошибок связано с аппаратурой, компилятором или ОС.

Если вы не знаете, почему это работает, вероятно, оно и не работает на самом деле.

Не спешите и не компилируйте программу, пока не будете уверены, что она верна.

Установите наивысший уровень предупреждений компилятора.

Применяйте проверяющие средства.

Убедитесь, что каждая строка выполняется так, как вы ожидаете.

Леса – код, поддерживающий методы при тестировании и не включаемый в конечный продукт.

Полное перепроектирование нестабильного метода полностью оправданно.

Если качество метода неудовлетворительное, вернитесь к псевдокоду.

Рекурсивное применение псевдокода – разбиение метода на более мелкие при необходимости.

Комментарии должны быть адекватными и полезными.

Неявное объявление переменных – одна из самых опасных возможностей языка.

Объявляйте все переменные.

Отключите неявные объявления.

Используйте конвенции именования.

Проверяйте имена переменных.

Инициализируйте каждую переменную при ее объявлении.

Инициализируйте каждую переменную там, где она используется в первый раз (если нельзя инициализировать при объявлении).

В идеальном случае сразу объявляйте и определяйте каждую переменную непосредственно перед первым обращением к ней.

Объявляйте переменные по мере возможности как `const`.

Не забывайте обнулять счетчики и аккумуляторы.

Специализируйте данные-члены класса в его конструкторе.

Проверяйте необходимость повторной инициализации.

Область видимости – фрагмент программы, в котором переменная известна и может быть использована.

Локализируйте обращения к переменным (уменьшать интервал).

Делайте время жизни переменных как можно короче (измеряется в строках).

Избегайте глобальных переменных из-за большого времени жизни.

Инициализируйте переменные, используемые в цикле, непосредственно перед циклом, а не в начале метода, содержащего цикл.

Не присваивайте переменной значение вплоть до его использования.

Группируйте связанные команды.

Разбивайте группы связанных команд на отдельные методы.

Начинайте с самой ограниченной области видимости и расширяйте ее только при необходимости.

Локальная область видимости способствует интеллектуальной управляемости.

Персистентность характеризует длительность существования данных.

Время связывания – момент, когда переменная и ее значение связываются вместе.

Так как эффективность программирования зависит от минимизации сложности, опытный программист будет обеспечивать такой уровень гибкости, какой нужен для удовлетворения требований, но не более того.

Последовательные данные соответствуют последовательности команд.

Селективные данные соответствуют операторам `if` и `case`.

Итеративные данные соответствуют циклам.

Используйте каждую переменную только с одной целью.

Избегайте переменных, имеющих скрытый смысл (избегайте гибридного сопряжения).

Убеждайтесь в том, что используются все объявленные переменные.

Имя переменной должно полно и точно описывать сущность, представляемую переменной.

Имена должны быть максимально конкретны.

Имя переменной описывает проблему, а не ее решение.

Отладка программы требует меньше усилий, если имена переменных состоят в среднем из 10–16 символов.

Более длинные имена лучше присваивать редко используемым или глобальным переменным, а более короткие – локальным переменным или переменным, вызываемым в циклах.

Дополняйте имена, относящиеся к глобальному пространству имен, спецификаторами.

Имена вычисляемых значений дополнять спецификатором из Total, Sum, Average, Max, Min, Record, String, Pointer в конце имени.

В именах лучше использовать антонимы: begin/end, first/ last, locked/unlocked, min/max, next/previous, old/new, opened/ closed, visible/invisible, source/target, source/destination, up/ down.

Индексы циклов – i, j, k.

Если код часто изменяется, модернизируется, копируется, лучше делать осмысленные имена индексов.

Имя флага не должно включать фрагмент flag, потому что он ничего не говорит о сути флага.

Значения флагов лучше сравнивать со значениями перечислений, именованных констант или глобальных переменных, выступающих в роли именованных констант.

Использование временных переменных говорит о том, что программист еще не полностью понял проблему.

Временным переменным давать точное описание.

Типичные имена булевых переменных: признак завершения – done, признак ошибки – error, признак обнаружения – found, признак успешного завершения – ok или success.

Присваивайте булевым переменным имена, подразумевающие значение true или false.

Используйте утвердительные имена булевых переменных.

Имя константы должно характеризовать абстрактную сущность, представляемую константой, а не конкретное значение.

Обязательно используйте конвенции именования, так как это всегда выгодно.

Советы по созданию конвенции:

Проведите различие между именами переменных и именами методов.

Проведите различие между классами и объектами.

Идентифицируйте глобальные переменные, переменные члены, определения типов, именованные константы, элементы перечислений, неизменяемые параметры.

Формируйте имена так, чтобы их было легко читать.

Сокращение имен стало пережитком.

Не сокращайте слова только на один символ.

Всегда используйте один и тот же вариант сокращения.

Сокращайте имена так, чтобы их можно было произнести.

Избегайте комбинаций, допускающих неверное прочтение или произношение имен.

Обращайтесь к словарю для разрешения конфликтов имен.

Документируйте очень короткие имена прямо в коде при помощи таблиц.

Указывайте все сокращения в проектном документе «Стандартные аббревиатуры».

Помните, что имена создаются в первую очередь для программистов, читающих код.

Избегайте обманчивых имен или аббревиатур.

Избегайте имен, имеющих похожие значения.

Избегайте переменных, имеющих разную суть, но похожие имена.

Избегайте имен, имеющих похожее звучание, таких как `wgar` и `gar`.

Избегайте имен, включающих цифры.

Избегайте орфографических ошибок.

Избегайте слов, при написании которых люди часто допускают ошибки.

Проводите различие между именами не только по регистру букв.

Избегайте смешения естественных языков.

Избегайте имен стандартных типов, переменных и методов.

Не используйте имена, которые совершенно не связаны с тем, что представляют переменные.

Избегайте имен, содержащих символы, которые можно спутать с другими символами.

Избегайте «магических чисел».

Используйте в программе как константы только 0 и 1, а любые другие числа определите как литералы с понятными именами.

Пишите код, предупреждающий появление ошибки деления на 0.

Выполняйте преобразования типов понятно.

Избегайте сравнений разных типов.

Обращайте внимание на предупреждения вашего компилятора.

Проверяйте целочисленность операций деления.

Проверяйте переполнение целых чисел.

Проверяйте на переполнение промежуточные результаты.

Избегайте сложения и вычитания слишком разных по размеру чисел с плавающей запятой.

Избегайте сравнений на равенство чисел с плавающей запятой.

Предупреждайте ошибки округления.

Проверяйте поддержку специальных типов данных в языке и дополнительных библиотеках.

Избегайте магических символов и строк.

Следите за ошибками завышения/занижения на единицу индексов.

Узнайте, как ваш язык и система поддерживают Unicode.

Разработайте стратегию интернационализации/локализации в ранний период жизни программы.

Если вам известно, что нужно поддерживать только один алфавит, рассмотрите вариант использования набора символов ISO8859.

Если вам необходимо поддерживать несколько языков, используйте Unicode.

Выберите целостную стратегию преобразования строковых типов.

Используйте логические переменные для документирования программы.

Используйте логические переменные для упрощения сложных условий.

Используйте перечислимые типы для читабельности.

Используйте перечислимые типы для надежности.

Используйте перечислимые типы для модифицируемости.

Используйте перечислимые типы как альтернативу логическим переменным.

Проверяйте некорректные значения в case с перечислимым типом.

Настройте первый и последний элемент перечислимого типа для использования в качестве границ циклов.

Зарезервируйте первый элемент перечислимого типа как недопустимый.

Точно укажите в стандартах кодирования для проекта, как должны использоваться первый и последний элементы, и неукоснительно придерживайтесь этого.

Помните о подводных камнях в присваивании явных значений элементам перечисления.

Если в языке нет перечислимых типов, их можно имитировать, используя глобальные переменные или классы.

Используйте именованные константы в объявлениях данных.

Избегайте литеральных значений, даже «безопасных».

Если в языке их нет, имитируйте именованные константы с помощью переменных или классов правильной области видимости.

Не используйте для представления одной сущности именованные константы в одном месте и литералы в другом.

Убедитесь, что все значения индексов массива не выходят за его границы.

Обдумайте применение контейнеров вместо массивов или рассматривайте массивы как последовательные структуры (наборов, стеков, очередей и т. п.).

Проверяйте конечные точки массивов.

В многомерном массиве убедитесь, что его индексы используются в правильном порядке.

Остерегайтесь пересечения индексов – перемены мест индексов.

Принципы создания собственных типов:

Создавайте типы с именами, отражающими их функциональность.

Преимущество создания собственных типов в появлении слоя, скрывающего язык.

Избегайте предопределенных типов.

Не переопределяйте предопределенные типы.

Определите подстановки для переносимости.

Рассмотрите вопрос создания класса вместо использования typedef.

Используйте структуры для упрощения списка параметров.

Между методами должна передаваться только та информация, которую необходимо знать.

Используйте структуры для упрощения сопровождения.

Используйте технологию, не основанную на указателях.

Избегайте преднамеренных изменений глобальных данных.

Глобальные данные должны сохранять свои значения, даже если будет запущено несколько копий программы.

Глобальные данные затрудняют повторное использование кода.

Глобальные данные приводят к неопределенному порядку инициализации.

Глобальные данные привносят нарушение модульности и интеллектуальной управляемости.

Глобальные данные подходят для хранения глобальных значений.

Глобальные данные подходят для эмуляции именованных констант, если они не поддерживаются.

Глобальные данные подходят для эмуляции перечислимых типов.

Глобальные данные можно использовать для оптимизации обращений к часто используемым данным.

Применение глобальных переменных позволяет избежать бродячие данные.

Начните с объявления всех переменных локальными и делайте их глобальными только по необходимости.

Различайте глобальные переменные и переменные-члены класса. Используйте методы доступа.

Требуйте, чтобы в свойствах весь код обращался к данным через методы доступа.

Не валите все глобальные данные в одну кучу.

Управляйте доступом к глобальным переменным с помощью блокировок.

Встройте уровень абстракции в методы доступа (не используйте структуры напрямую).

Выполняйте доступ к данным на одном и том же уровне абстракции.

Разработайте соглашения по именованию, которые сделают глобальные переменные очевидными.

Создайте хорошо аннотированный список всех глобальных переменных.

Не храните промежуточных результатов в глобальных переменных.

Не считайте, что вы не используете глобальные переменные, поместив все данные в чудовищный объект и передавая его всюду.

Операторы:

Факт зависимости одного выражения от другого должен быть понятен из имен методов.

Организируйте код так, чтобы зависимости между операторами были очевидными.

Называйте методы так, чтобы зависимости были очевидными.

Используйте параметры методов, чтобы сделать зависимости очевидными.

Документируйте неявные зависимости с помощью комментариев.

Проверяйте зависимости с помощью утверждений или кода обработки ошибок.

Располагайте взаимосвязанные действия вместе.

При написании if-выражений:

Сначала напишите код номинального хода алгоритма, затем опишите исключительные ситуации.

Убедитесь, что при сравнении на равенство ветвление корректно.

Размещайте нормальный вариант после if, а не после else.

Размещайте осмысленные выражения после оператора if.

Рассмотрите вопрос использования блока else (в 50–80% случаев использования if следовало применить и else).

Проверяйте корректность выражения else.

Проверяйте возможную перестановку блоков if и else.

Упрощайте сложные проверки с помощью вызовов логических функций.

Размещайте наиболее вероятные варианты раньше остальных.

Убедитесь, что учтены все варианты.

Замените последовательности if-then-else другими конструкциями, которые поддерживает ваш язык программирования.

В case:

- упорядочивайте варианты case по алфавиту или численно;

- поместите правильный вариант case первым;

- отсортируйте варианты по частоте.

Сделайте обработку каждого варианта case простой.

Не конструируйте искусственные переменные с целью получить возможность использовать оператор case.

Используйте вариант по умолчанию только для обработки настоящих значений по умолчанию.

Используйте вариант по умолчанию для выявления ошибок.

Старайтесь не писать код, проваливающийся сквозь блоки оператора case.

Минимизируйте число факторов, влияющих на цикл.

Вынесите за пределы цикла все управление, какое только можно.

Размещайте вход в цикл только в одном месте.

Размещайте инициализационный код непосредственно перед циклом.

Используйте while (true) для бесконечных циклов.

Предпочитайте циклы for, если они применимы.

Не используйте цикл for, если цикл while подходит больше.

Используйте {} для обрамления выражений в цикле.

Избегайте пустых циклов.

Располагайте служебные операции либо в начале, либо в конце цикла.

Заставьте каждый цикл выполнять только одну функцию.

При завершении цикла убедитесь, что выполнение цикла закончилось.

Сделайте условие завершения цикла очевидным.

Не играйте с индексом цикла for для завершения цикла.

Избегайте писать код, зависящий от последнего значения индекса цикла.

Рассмотрите использование счетчиков безопасности, чтобы определить, не слишком ли много раз выполняется цикл.

Рассмотрите использование операторов break вместо логических флагов в цикле while.

Остерегайтесь цикла с множеством операторов break, разбросанных по всему коду.

Используйте continue для проверок в начале цикла.

Используйте структуру break с метками, если ваш язык ее поддерживает.

Используйте операторы break и continue очень осторожно.

Используйте порядковые или перечислимые типы для границ массивов и циклов.

Используйте смысловые имена переменных, чтобы сделать вложенные циклы читабельными.

Используйте смысловые имена во избежание пересечения индексов.

Ограничивайте видимость переменных-индексов цикла самим циклом.

Делайте циклы достаточно короткими, чтобы их можно было увидеть сразу целиком.

Ограничивайте вложенность тремя уровнями.

Выделяйте внутреннюю часть длинных циклов в отдельные методы.

Делайте длинные циклы особенно ясными.

Простое создание цикла – изнутри наружу.

Используйте return, если это повышает читабельность.

Упрощайте сложную обработку ошибок с помощью сторожевых операторов (досрочного return).

Минимизируйте число возвратов из каждого метода.

Убедитесь, что рекурсия остановится.

Предотвращайте бесконечную рекурсию с помощью счетчиков безопасности.

Ограничьте рекурсию одним методом.

Следите за стеком при использовании рекурсии.

Не используйте рекурсию для факториалов и чисел Фибоначчи.

Лучше обходиться без goto.

Код с goto переписать с помощью вложенных if.

Код с goto переписать с использованием статусной переменной.

Табличный метод – это схема, позволяющая искать информацию в таблице, а не использовать для этого логические выражения.

Таблица с прямым доступом позволяет обращаться изначально напрямую (есть еще индексированный и ступенчатый доступ).

Используйте неявное сравнение логических величин с true или false.

Разбивайте сложные проверки на части с помощью новых логических переменных.

Размещайте сложные выражения в логических функциях.

Используйте таблицы решений для замены сложных условий.

В операторах if заменяйте негативные выражения позитивными, меняя местами блоки if и else.

Применяйте теоремы Деморгана для упрощения логических проверок с отрицаниями.

Используйте скобки для явного задания порядка вычисления.

Используйте простой метод подсчета для проверки симметричности скобок $((+1,) - -1)$.

Заклучайте в скобки логическое выражение целиком.

Организируйте числовые условия так, чтобы они соответствовали порядку точек на числовой прямой.

Неявно сравнивайте логические переменные.

Сравнивайте числа с 0.

Сравнивайте указатели с null.

В C-подобных языках помещайте константы с левой стороны сравнений.

Учитывайте разницу между `a==b` и `a.equals(b)` (Java).

Пишите обе скобки блока одновременно.

Всегда используйте скобки для условных операторов (для пояснения).

Привлекайте внимание к нужным выражениям.

Создайте для пустых выражений макрос препроцессора или встроенную функцию `DoNothing()`.

Подумайте, не будет ли код яснее с непустым телом цикла.

Не используйте больше 3 уровней вложенности if.

Упростите вложенные if с помощью повторной проверки части условия.

Упростите вложенные if с помощью блока с выходом `do{...break...} while (false);`.

Преобразуйте вложенные if в набор if-then-else.

Преобразуйте вложенные if в оператор case.

Факторизуйте глубоковложенный код в отдельный метод.

Используйте более объекто-ориентированный подход.

Перепроектируйте глубоко вложенный код.

Основной тезис структурного программирования: любая управляющая логика программы может быть реализована с помощью последовательности, выбора и итерации.

Методики поиска дефектов лучше применять в комбинации.

Обзор кода эффективнее тестирования.

Ошибки в требованиях или архитектуре обычно имеют более широкие следствия, чем ошибки конструирования.

Повышение качества системы снижает расходы на ее разработку.

Средняя производительность труда программистов эквивалентна 10–50 строкам кода на одного человека в день.

Устранение дефектов – самый дорогой и длительный этап разработки ПО.

На создание ПО без дефектов не всегда уходит больше времени, чем на написание ПО с дефектами.

Разработчики допускают в среднем от 1 до 3 дефектов в час при проектировании и от 5 до 8 дефектов в час при кодировании.

Каждый час инспекции кода предотвращает около 100 часов аналогичной работы (тестирования и исправления дефектов).

Поддерживайте парное программирование стандартами кодирования.

Не позволяйте парному программированию превратиться в наблюдение.

Не используйте парное программирование для реализации простых фрагментов.

Регулярно меняйте состав пар и назначаемые парам задачи.

Объединяйте в пару людей, предпочитающих одинаковый темп работы.

Убедитесь, что оба члена пары видят экран.

Не объединяйте в пары людей, которые не нравятся друг другу.

Не составляйте пару из людей, которые ранее не программировали в паре.

Назначьте лидера группы.

Инспектор может проанализировать за час около 500 строк.

Тестирование – самая популярная методика повышения качества.

Тестирование – средство обнаружения ошибок.

Отладка – средство поиска и устранения причин уже обнаруженных ошибок.

Тестированию, выполняемому разработчиками, следует посвящать от 8% до 25% общего времени работы над проектом.

Искусство тестирования заключается в выборе тестов, способных обеспечить максимальную вероятность обнаружения ошибок.

Структурированное базисное тестирование – протестировать каждый оператор программы хотя бы раз.

Проверяйте код тестов.

Планируйте тестирование программы так же, как и ее разработку.

Храните тесты.

Встраивайте блочные тесты в среду тестирования.

Используйте генераторы случайных данных.

Изучите программу, над которой работаете.

Изучите собственные ошибки.

Изучите качество своего кода с точки зрения кого-то, кому придется читать его.

Изучите используемые способы решения проблем.

Изучите используемые способы исправления дефектов.

Формулируя гипотезу, используйте все имеющиеся данные.

Детализируйте тесты, приводящие к ошибке.

Проверяйте код при помощи блочных тестов.

Используйте разные инструменты.

Воспроизведите ошибку несколькими способами.

Генерируйте больше данных для формулирования большего числа гипотез.

Используйте «мозговой штурм» для построения нескольких гипотез.

Составьте список подходов, которые стоит попробовать.

Сохраните подозрительную область кода.

С подозрением относитесь к классам, которые содержали дефекты ранее.

Проверьте код, который был изменен недавно.

Расширьте подозрительный фрагмент кода.

Выполняйте интеграцию инкрементно.

Проверяйте наличие распространенных дефектов.

Обсудите проблему с кем-то другим.

Отдохните от проблемы.

Установите лимит времени для быстрой и грязной отладки.

Составьте список методик грубой силы.

Не полагайтесь на номера строк в сообщениях компилятора.

Не доверяйте сообщениям компилятора.

Не доверяйте второму сообщению компилятора.

Разделяйте программу на части.

Грамотно ищите неверно размещенные комментарии и кавычки.

Прежде чем браться за решение проблемы, поймите ее.

Подтвердите диагноз проблемы.

Расслабьтесь (не поддавайтесь соблазну сэкономить время).

Сохраняйте первоначальный исходный код.

Устраняйте проблему, а не ее симптомы.

Изменяйте код только при наличии веских оснований.

Вносите в код по одному изменению.

Добавляйте в набор тестов блочные тесты, приводящие к проявлению имеющихся дефектов.

Поищите похожие дефекты.

Выбирайте во время конструирования имени, ясно отличающиеся от других имен.

Используйте утилиты сравнения исходного кода.

Задайте в компиляторе максимально строгий уровень диагностики и устраняйте все ошибки и предупреждения.

Рассматривайте предупреждения как ошибки.

Стандартизируйте параметры компилятора в масштабе всего проекта.

Используйте утилиты расширенной проверки синтаксиса и логики.

Профилируйте код для поиска ошибок.

Даже в хорошо управляемых проектах требования изменяются на 1–4% в месяц.

Современные подходы снижают предсказуемость кодирования.

Факторинг – максимальная декомпозиция программы на составляющие части.

Причины выполнения рефакторинга:

Код повторяется.

Метод слишком велик.

Цикл слишком велик или слишком глубоко вложен в другие циклы.

Класс имеет плохую связность.

Интерфейс класса не формирует согласованную абстракцию.

Метод принимает слишком много параметров.

Отдельные части класса изменяются независимо от других частей.

При изменении программы требуется параллельно изменять несколько классов.

Вам приходится параллельно изменять несколько иерархий наследования.

Вам приходится параллельно изменять несколько блоков case.

Родственные элементы данных, используемые вместе, не организованы в классы.

Метод использует больше элементов другого класса, чем своего собственного.

Элементарный тип данных перегружен.
Класс имеет слишком ограниченную функциональность.
По цепи методов передаются бродячие данные.
Объект-посредник ничего не делает.
Один класс слишком много знает о другом классе.
Метод имеет неудачное имя.

Данные-члены сделаны открытыми.

Подкласс использует только малую долю методов своих предков.

Сложный код объясняется при помощи комментариев.

Код содержит глобальные переменные.

Перед вызовом метода выполняется подготовительный код (после вызова метода выполняется код уборки).

Программа содержит код, который может когда-нибудь понадобиться.

Рефакторинг на уровне данных:

Замена магического числа на именованную константу.

Присвоение переменной более ясного или информативного имени.

Встраивание выражения в код.

Замена выражения на вызов метода.

Введение промежуточной переменной.

Преобразование многоцелевой переменной в несколько одноцелевых переменных.

Использование локальной переменной вместо параметра.

Преобразование элементарного типа данных в класс.

Преобразование набора кодов в класс или перечисление.

Преобразование набора кодов в класс, имеющий производные классы.

Преобразование массива в класс.

Инкапсуляция набора.

Замена традиционной записи на класс данных.

Рефакторинг на уровне отдельных операторов:

Декомпозиция логического выражения.

Вынесение сложного логического выражения в грамотно названную булеву функцию.

Консолидация фрагментов, повторяющихся в разных частях условного оператора.

Использование оператора break или return вместо управляющей переменной цикла.

Возврат из метода сразу после получения ответа вместо установки возвращаемого значения внутри вложенных операторов if-then-else.

Замена условных операторов (обычно многочисленных блоков case) на вызов полиморфного метода.

Создание и использование «пустых» объектов вместо того, чтобы проверять, равно ли значение null.

- Извлечение метода из другого метода.
- Встраивание кода метода.
- Преобразование объемного метода в класс.
- Замена сложного алгоритма на простой.
- Добавление параметра.
- Удаление параметра.
- Отделение операций запроса данных от операций изменения данных.
- Объединение похожих методов путем их параметризации.
- Разделение метода, поведение которого зависит от полученных параметров.
- Передача в метод целого объекта вместо отдельных полей.
- Передача в метод отдельных полей вместо целого объекта.
- Инкапсуляция нисходящего приведения типов.
- Рефакторинг реализации классов:
- Замена объектов-значений на объекты-ссылки.
- Замена объектов-ссылок на объекты-значения.
- Замена виртуальных методов на инициализацию данных.
- Изменение положения методов-членов или данных-членов в иерархии наследования.
- Перемещение специализированного кода в подкласс.
- Объединение похожего кода и его перемещение в суперкласс.
- Рефакторинг интерфейсов классов:
- Перемещение метода в другой класс.
- Разделение одного класса на несколько.
- Удаление класса.
- Соккрытие делегата.
- Удаление посредника.
- Замена наследования на делегирование.
- Замена делегирования на наследование.
- Создание внешнего метода.
- Создание класса-расширения.
- Инкапсуляция открытой переменной-члена.
- Удаление методов установки значений неизменяемых полей.
- Соккрытие методов, которые не следует вызывать извне класса.
- Инкапсуляция неиспользуемых методов.
- Объединение суперкласса и подкласса, имеющих очень похожую реализацию.
- Рефакторинг на уровне системы:

Создание эталонного источника данных, которые вы не можете контролировать.

Изменение однонаправленной связи между классами на двунаправленную.

Изменение двунаправленной связи между классами на однонаправленную.

Предоставление фабричного метода вместо простого конструктора.

Замена кодов ошибок на исключения или наоборот.

Сохраняйте первоначальный код.

Стремитесь ограничить объем отдельных видов рефакторинга.

Выполняйте отдельные виды рефакторинга по одному за раз.

Составьте список действий, которые вы собираетесь предпринять.

Составьте и поддерживайте список видов рефакторинга, которые следует выполнить позже.

Часто создавайте контрольные точки.

Используйте предупреждения компилятора.

Выполняйте регрессивное тестирование.

Создавайте дополнительные тесты.

Выполняйте обзоры изменений.

Изменяйте подход в зависимости от рискованности рефакторинга.

Не рассматривайте рефакторинг как оправдание написания плохого кода с намерением исправить его позднее.

Не рассматривайте рефакторинг как способ, позволяющий избежать переписывания кода.

Тратьте время на 20% видов рефакторинга, обеспечивающих 80% выгоды.

Выполняйте рефакторинг при создании новых методов.

Выполняйте рефактинги при создании новых классов.

Выполняйте рефакторинг при исправлении дефектов.

Выполняйте рефакторинг модулей, подверженных ошибкам.

Выполняйте рефакторинг сложных модулей.

При сопровождении программы улучшайте фрагменты, к которым прикасаетесь.

Определите интерфейс между аккуратным и безобразным кодом и переместите безобразный код на другую сторону этого интерфейса.

Простое задание явных целей повышает вероятность их достижения.

На 20% методов программы приходится 80% времени ее выполнения.

Сокращение числа строк высокоуровневого кода не повышает быстродействие и не уменьшает объем итогового машинного кода.

Без измерения производительности вы никак не сможете точно узнать, помогли ваши изменения программе или навредили.

Методики, повышающие производительность в одной среде, могут снижать ее в других.

До создания полностью работоспособной программы найти узкие места в коде почти невозможно.

Концентрация на производительности во время первоначальной разработки отвлекает от достижения других целей.

Корректность важнее быстродействия.

Частые причины снижения эффективности:

Операции ввода/вывода.

Замещение страниц памяти.

Системные вызовы.

Интерпретируемые языки.

Ошибки.

Повышение быстродействия исходит из замены дорогой операции на более дешевую.

Главная проблема оптимизации кода: результат любого отдельного вида оптимизации непредсказуем.

Замыкание цикла – принятие решения внутри цикла при каждой его итерации.

Разомкнутый цикл – принятие решения вне цикла.

Если два цикла работают с одним набором элементов, можно выполнить их объединение.

После частичного развертывания цикла при каждой его итерации обрабатывается не один случай, а два и более.

Вкладывайте более ресурсоемкий цикл в менее ресурсоемкий.

Минимизируйте число обращений к массивам.

Даже слепые белки иногда наталкиваются на орехи.

Назначьте двух человек на каждую часть проекта.

Рецензируйте каждую строку кода.

Введите процедуру подписания кода.

Распространяйте для ознакомления хорошие примеры кода.

Подчеркивайте, что код – это общее имущество.

Награждайте за хороший код.

«Я должен быть в состоянии прочесть и понять любой код, написанный в проекте».

Следуйте систематической процедуре контроля изменений.

Обрабатывайте затраты на каждое изменение.

Оценивайте затраты на каждое изменение.

Относитесь с подозрением к изменениям большого объема.

Учредите комитет контроля изменений.

Соблюдайте бюрократические процедуры, но не позволяйте страху перед бюрократией препятствовать эффективному контролю изменений.

Оценка графика конструирования:

Определите цели.

Выделите время для оценки.

Выясните требования к программе.

Делайте оценки на низком уровне детализации.

Используйте несколько способов оценки и сравнивайте полученные результаты.

Периодически делайте повторную оценку.

Храните сведения об опыте проектов в вашей организации и используйте их для оценки времени, необходимого будущим проектам.

Наибольшее влияние на график программного проекта оказывает размер создаваемой программы.

Если вы отстаете:

Надеяться, что вы сможете наверстать упущенное.

Задержки и отклонения от графика обычно увеличиваются по мере приближения к концу проекта.

Увеличить команду (если есть независимые задачи).

Сократить проект.

При первоначальном планировании продукта разделите его возможности на категории «должны быть», «хорошо бы сделать», «необязательные».

Причины, по которым стоит проводить измерение проекта:

Для любого атрибута проекта существует возможность его измерения, что в любом случае не означает отказа от его измерения.

Отдавайте себе отчет о побочных эффектах измерения.

Возражать против измерений означает утверждать, что лучше не знать о том, что на самом деле происходит.

80% работы выполняют 20% сотрудников.

Если вы хотите контролировать стиль программиста:

Вы вторгаетесь в область, требующую деликатного обращения.

Из уважения к теме используйте термины «предложения» и «советы».

Старайтесь обходить стороной спорные вопросы, предпочитая давать явные поручения.

Предложите программистам выработать собственные стандарты.

Просвещайте менеджеров.

Писать и тестировать маленькие участки программы, а затем комбинировать эти кусочки друг с другом по одному.

Преимущества этой инкрементной итерации:

Ошибки можно легко обнаружить.

В таком проекте система раньше становится работоспособной.

Вы получаете улучшенный мониторинг состояния.

Вы улучшите отношения с заказчиком.

Системные модули тестируются гораздо полнее.

Вы можете создать систему за более короткое время.

При нисходящей интеграции вы создаете те классы, которые находятся на вершине иерархии, первыми, а те, что внизу, – последними.

Систему можно разбить на вертикальные слои.

При восходящей интеграции вы пишете и интегрируете сначала классы, находящиеся внизу иерархии.

При риск-ориентированной интеграции вы решаете, какие части системы будут самыми трудными, и реализуете их первыми.

Еще один подход – интеграция одной функции в каждый момент времени.

Создавайте сборку ежедневно.

Проверяйте правильность сборок.

Выполняйте дымовые тесты ежедневно.

Поддерживайте актуальность дымового теста.

Автоматизируйте ежедневную сборку и дымовой тест.

Организируйте группу, отвечающую за сборки.

Вносите исправления в сборку, только когда имеет смысл это делать, но не откладываете внесение исправлений надолго.

Требуйте, чтобы разработчики проводили дымовое тестирование своего кода перед его добавлением в систему.

Создайте область промежуточного хранения кода, который следует добавить к сборке.

Назначьте наказание за нарушение сборки.

Выпускайте сборки по утрам.

Создавайте сборку и проводите дымовой тест даже в экстремальных условиях.

Под «непрерывной» понимается «по крайней мере, ежедневная» интеграция.

До 40% времени программист тратит на редактирование исходного кода.

Передовой набор инструментов позволяет повысить производительность более чем на 50%.

20% инструментария используются в 80% случаев.

Хорошее визуальное форматирование показывает логическую структуру программы.

Абзац кода должен содержать только взаимосвязанные операторы, выполняющие одно задание.

Начало нового абзаца в коде нужно указывать с помощью пустой строки.

Оптимальное число пустых строк в программе – 16%.

Операторы выделяются отступами, когда они следуют после выражения, от которого логически зависят.

Оптимальными являются отступы из 2–4 пробелов.

Формируйте блоки из 1 оператора единообразно.

В сложных выражениях размещайте каждое условие на отдельной строке.

Избегайте операторов `goto`.

Не используйте форматирование в конце строки в виде исключения для операторов `case`.

Используйте пробелы, чтобы сделать читаемыми логические выражения.

Используйте пробелы, чтобы сделать читаемыми обращения к массиву.

Используйте пробелы, чтобы сделать читаемыми аргументы методов.

Сделайте так, чтобы незавершенность выражения была очевидна.

Располагайте сильно связанные элементы вместе.

При переносе строк в вызове метода используйте отступ стандартного размера.

Упростите поиск конца строки с продолжением.

При переносе строк в управляющем выражении делайте отступ стандартного размера.

Не выравнивайте правые части выражений присваивания.

При переносе строк в выражениях присваивания применяйте отступы стандартного размера.

Располагайте каждое объявление данных в отдельной строке.

Объявляйте переменные рядом с местом их первого использования.

Разумно упорядочивайте объявления.

Делайте в комментарии такой же отступ, как и в соответствующем ему коде.

Отделяйте каждый комментарий хотя бы одной пустой строкой.

Используйте пустые строки для разделения составных частей метода.

Используйте стандартный отступ для аргументов метода.

Если файл содержит более 1 класса, четко определяйте границы каждого класса.

Помещайте каждый класс в отдельный файл.

Называйте файл в соответствии с именем класса.

Отделяйте методы друг от друга с помощью хотя бы 2 пустых строк.

Упорядочивайте методы по алфавиту.

Используйте стиль комментирования, который легко поддерживать.

Используйте содержательные комментарии.

Избегайте комментариев, высосанных из пальца.

Не используйте комментарии в концах строк, относящиеся к нескольким строкам кода.

Используйте комментарии в концах строк для пояснения объявлений данных.

Не используйте комментарии в концах строк для вставки пометок во время сопровождения ПО.

Используйте комментарии в концах строк для обозначения концов блоков.

Описывайте цель блока кода, следующего за комментарием.

Во время документирования сосредоточьтесь на самом коде.

Придумывая комментарий абзаца, стремитесь ответить на вопрос «почему», а не «как».

Используйте комментарии для подготовки читателя кода к последующей информации.

Не размножайте комментарии без необходимости.

Документируйте сюрпризы.

Избегайте сокращений.

Проведите различие между общими и детальными комментариями.

Комментируйте все, что имеет отношение к ошибкам или недокументированным возможностям языка или среды.

Указывайте в комментариях единицы измерения численных величин.

Указывайте в комментариях диапазоны допустимых значений численных величин.

Комментируйте смысл закодированных значений.

Комментируйте ограничения входных данных.

Документируйте флаги до уровня отдельных битов.

Включайте в комментарии, относящиеся к переменной, имя переменной.

Документируйте глобальные данные.

Пишите комментарий перед каждым оператором if, блоком case, циклом или группой операторов.

Комментируйте завершение каждой управляющей структуры.

Рассматривайте комментарии в концах циклов как предупреждения о сложности кода.

Располагайте комментарии близко к описываемому ими коду.

Описывайте каждый метод одним-двумя предложениями перед началом метода.

Документируйте параметры в местах их объявления.

Используйте утилиты документирования кода.

Проведите различие между входными и выходными данными.

Документируйте выраженные в интерфейсе предположения.

Комментируйте ограничения методов.

Документируйте глобальные результаты выполнения метода.

Документируйте источники используемых алгоритмов.

Используйте комментарии для маркирования частей программы.

Опишите подход к проектированию класса.

Опишите ограничения класса, предположения о его использовании и так далее.

Прокомментируйте интерфейс класса.

Не документируйте в интерфейсе класса детали реализации.

Опишите назначение и содержание каждого файла.

Укажите в блочном комментарии свои имя/фамилию, адрес электронной почты и номер телефона.

Включите в файл тег версии.

Включите в блочный комментарий юридическую информацию.

Присвойте файлу имя, характеризующее его содержание.

Лучшие программисты создают программы в 10 раз быстрее коллег.

Изучите процесс разработки.

Экспериментируйте.

Читайте о решении проблем.

Анализируйте и планируйте, прежде чем действовать.

Изучайте успешные проекты.

Читайте.

Читайте другие книги и периодические издания.

Общайтесь с единомышленниками.

Постоянно стремитесь к профессиональному развитию.

Математизация автороведческой экспертизы (А. С. Лот)

В древние времена письменность была доступна только элите, имевшей значительное влияние среди остальных людей. Глиняные дощечки и папирус служили средством передачи информации. С появлением книгопечатания все больше читателей появлялось у огромного числа авторов. Сегодняшний человек не мыслит себя без средств массовой информации и книг. С ростом популярности авторов увеличивалось количество их подражателей. Стало подвергаться сомнению авторство старинных текстов. В связи с этим возникла потребность в автороведческой экспертизе, которой и посвящена эта работа. Задача установления авторства определяет необходимость построения алгоритма решения в виде последовательности действий, производимых над текстами. Этот тривиальный на первый взгляд подход требует от разработчика алгоритма глубочайшей осмысленности операций при обработке данных авторов: только качественное решение дает гарантированный результат. Интуитивно задача идентификации авторства имеет бесчисленное количество решений. Несомненно, язык автора и состояние самого автора меняются от произведения к произведению. Поэтому в алгоритм решения можно внести допущение на ошибку, которое помогло бы нивелировать вышеупомянутый эффект. По Маркову, авторским текстам присуще наличие шумов и языковых особенностей. Вместе с тем произведение автора содержит, собственно, авторские компоненты, выделить которые достаточно для определения всех текстов этого автора среди неизвестных трудов. Рассматривая текст произведения на временной оси, легко отметить, что время, затрачиваемое на написание какой-либо логически завершенной части текста автором, будет пропорционально его длине как оценке затраченного труда. Когда мы рассматриваем предпринимательскую способность как фактор, обуславливающий жанр, целевую аудиторию и прочие видимые невооруженным глазом любопытному ученому характеристики литературного произведения, то невольно обнаруживаем, что отчасти эта способность задает направление прочих затрат ради написания произведения: времени и писательского таланта. Таким образом, предпринимательская способность, будучи приложенной в сфере литературного авторства, во многом будет влиять на протяженность текста в единицах его длины. Не исключая из устремлений автора замысел просвещения, зафиксируем сходящие с его пера длины наименьших, логически завершенных и могущих быть в каком-то ракурсе полными смысла в конкретном произведении единиц текста произвольного его произведения, растянутого на временной оси, чтобы приблизиться к пониманию экспериментального подсчета рассматриваемой предпри-

нимательской способности, доля которой внесена в текст. За такую единицу текста примем одно его предложение как наиболее завершенную часть внутри самого крупного элемента исследуемого материала – текста произведения автора. Предпринимательская способность подчиняет себе писательский талант во времени, она формируется с течением жизни автора, дается с рождения и может утрачиваться. Поэтому категория завершенности должна в наибольшей мере быть присуща рассматриваемому произведению, закрывая его от дефектов – привнесений автора, обладающего сильно измененной предпринимательской способностью, а также других явлений, могущих исказить дух творчества, сопутствующий произведению. Признав счетной величиной предложение текста, необходимо избежать влияния длины всего текста произведения на научное распознавание духа творчества автора. Это связано с тем, что намеренно изменяемая длина текста могла бы легко ввести в заблуждение исследователя. Далеко не всегда автор может позволить себе облекать мысли в краткие, но емкие формулировки. Такое самоограничение творческой личности часто навлекается неготовностью читателя к некоторым сообщениям. Размышляя над подсчетом весомых характеристик автора, скрытых в тексте, можно разукрупнить предложение и подробнее рассмотреть его составляющие. Нам привычно мыслеформы облекать в слова, собирать из них предложения, используя связующее звено – русский язык. Слова, в свою очередь, язык разбивает на буквы и звуки, а среди прочих элементов текста остаются знаки пунктуации. Попробуем проследить связь предложения – единицы длины текста – с его конструктивными слагаемыми. Мы улавливаем смысл предложения, складывая и сопоставляя его слова – они могут передаваться нам как зрительный образ. Кооперативный эффект воспринимаемой последовательности букв слова, представленных кириллицей, легко понимается грамотным человеком. И такое представление очень удобно для работы со словом как объектом составным. Воспринимаемое в его звучании слово гораздо труднее поддается взятию как объект языка текста, да и современные ЭВМ гораздо легче позволяют работать с текстовой информацией, нежели звуком, полагаемым к дискретному счету. Таким образом, самым удобным способом манипуляции произведением признаем перебор букв его текста, а также знаков препинания, сохраняющих его структуру. Если бы анализ производился в речевом воспроизведении текста, то мы бы ушли от общеупотребительной формы представления художественных литературных произведений, дополнительно усложнив задачу построением системы распознавания речи. На данном этапе развития технологий и человека недопустимо пускаться в погоню за улучшением средств решения задачи в ущерб процессу ее решения.

Исходя из вышесказанного можно утверждать, что на пути к построению предложения автору предстоит преодолеть сложный процесс словотворчества. Слово есть в любом предложении. Оно выступает элементом протяженности его смысла, его глубины, значимости и отнесенности к объектам повествования. Слову присуща также дискретность. Всегда удастся подсчитать точное количество слов в предложении, которое не будет зависеть от каких-либо характеристик слов, кроме их наличия. Также немаловажно заметить, что наш исследуемый объект постоянен во времени его анализа, т.е. никакие измеряемые параметры не будут показывать различные значения с каждой новой попыткой подсчета по постоянному алгоритму. Не получится отыскать меньшую слова единицу текста, способную указать дух творчества его автора. Поэтому, опираясь на количество слов в его связи с предложением, которое бы не зависело от длины текста, можно охарактеризовать автора произведения как личность. Самый простой и эффективный способ получить такую величину – взять ее среднее арифметическое как непрерывную оценку. Среднее количество слов в предложении – действительная характеристика текста, указывающая связь художественного литературного произведения с духом творчества автора в его обусловленности предпринимательской способностью и писательским талантом. Предпринимательская способность здесь описывает необходимость употребления писательского таланта, в свою очередь содержащего богатство языка, меткость подстановки, словоохотливость, точность решения задач, перед которыми предпринимательская способность ставит автора и т. д. Зададимся целью установления наиболее точной единицы счета длины предложения как математической величины. Очевидно, что для лучшего понимания результата такая величина должна позволять натуральный счет и быть наименьшей относительно всего предложения из всех таких величин. Интуитивно такой мерой полагается одна буква предложения. Знаки препинания не будут существенно влиять на длину предложения, т. к. их количество намного меньше общего количества букв предложения. Буквы, в отличие от слов, которые можно принять единицей словарной, являются единицей значительно меньшего пространства – алфавитной. Мы не смогли бы отказаться от вычисления длин в словах в пользу побуквенного вычисления, т. к. несомый буквой в ее единичности смысл практически не воспринимаем. Вместе с тем хотелось бы учесть знаки препинания предложения в их связи с его конструкцией, обусловленной вводными составляющими, оборотами, перечислениями и т. п., несущей часть смысла целого предложения. Часто конструкции языка внутри предложения обрамляются в запятые – зачастую их несколько в одном предложении. Относительной величиной, подходящей в таком качестве,

является количество букв между запятыми в предложении. Причем опять берем среднее арифметическое этой величины. Наша привязанность к среднему арифметическому в математическом смысле объясняется следующим. Когда мы наблюдаем численные значения величины из анализа текста, то самой доступной к пониманию динамики составления текста является значение, которого стремится достичь эта величина. Такое значение в математической статистике называют математическое ожидание. Часто при решении задач оно бывает задано, однако в нашем случае сталкиваемся с неизвестным математическим ожиданием, т. к. параметры текста не были заданы изначально.

Как выполнить код на Assembler из кода C# под Linux (А. С. Лот)

Для работы понадобится ОС Linux и установленные в ней компиляторы gcc, g++. Проверить наличие компилятора можно командой `whereis`, а установить – командой `sudo apt install` в терминале. Имеется простейший код на GNU Assembler, возвращающий 1 (файл `asm. s`):

```
.text
.globl _new
_new:
mov $1,%rax
ret
```

Хотим запустить его в C# с помощью `PInvoke`. Для этого потребует-ся промежуточная динамическая библиотека на языке C с кодом (файл `my. c`):

```
#define EXPORT __attribute__ ((visibility ("default")))
EXPORT int foo (void);
int foo (void)
{
extern int _new ();
return _new ();
}
```

Эти два файла нужно собрать в `shared library` (файл `lib.so`) командой

```
gcc -shared -fpic -o lib.so my. c asm. s
```

Полученный файл `lib.so` нужно подложить в папку с исполняемым файлом результирующей программы на C#. У меня это подкаталог проекта `/bin/Debug/netcoreapp3.1/`. Код на C# для запуска функции `_new` выглядит так:

```
using System.Runtime.InteropServices;
class Program {
[DllImport("lib.so")] public static extern int foo ();
static void Main (string [] args)
{
int code = foo ();
System.Console. WriteLine (code);
}
}
```

Запускается он командой `dotnet run`, если у вас установлена исполняемая среда. NET Core. В консоли вы увидите 1. Чтобы добавить про-

слойку между C# и C в виде кода на C++, нужно добавить в проект файл my. cpp с кодом:

```
extern «C»{  
#define EXPORT __attribute__ ((visibility («default»)))  
EXPORT int foocpp (void)  
{  
extern int foo ();  
return foo ();  
}}
```

и заменить в коде C# название внешней функции foo на foocpp. Тогда изменится команда сборки и будет выглядеть так:

```
g++ -shared -fpic -o lib.so my. cpp -x c my. c -x assembler asm. s  
После чего запускаем dotnet run и видим 1 в консоли.
```

Синтаксический анализатор корректности текстовых арифметических выражений с использованием языка программирования Python

(А. С. Лот)

```
expressionChars = list(input («Enter expression:»))
isCorrectExpression = False
digits = {«0», «1», «2», «3», «4», «5», «6», «7», «8», «9»}
operations = {«/», «+», «-», «*», «^»}
state = 0;
count =
for char in expressionChars:
    if state == 0 and char in {«+», «-»}.union(digits): state = elif (state == 5
or state == 1 or state == 3) and char in digits:
        state = elif (state == 5 or state == 3 or state == 0) and char in {«»}:
            state = count+=elif state == 5 and char in {«+», «-»}:
                state = elif state == 1 and char in {«.»}:
                    state = elif (state == 1 or state == 4) and char in {«»}: state = count-
=elif (state == 1 or state == 4 or state == 2) and char in operations:
                state = elif (state == 6 or state == 2) and char in digits: state = if state
in {1, 4, 2} and count == 0: isCorrectExpression = True
    print (f«isCorrectExpression: {isCorrectExpression}»)
```

Допустимы отрицательные, положительные целые и рациональные числа, вычисления в круглых скобках, умножение, деление и возведение в степень (записывается знаком ^).

Что такое данные (А. С. Лот)

Толковый словарь русского языка Сергея Ожегова в 27-м издании приводит следующие трактовки слова «данные»: 1. Сведения, необходимые для какого-нибудь вывода, решения. 2. Основания для чего-нибудь, качества. В цифровую эпоху это слово обретает еще один – редуцированный смысл: данные – это число. Это – максимально конкретное определение, удобное для математизации любой сферы деятельности. Любые данные: изображения, видео, звук, текст, измерения. Любые данные сегодня способны быть представленными однозначно в цифровой форме, например, в распространенной системе счисления, – составляющей числа из нолей и единиц – двоичной. Из этой системы научными методами позволительно на сегодняшний день переводить в любую другую систему (или множество чисел) посредством алгоритмов. Отдельными алгоритмами двоичные данные могут переводиться из числа в текст, из текста в число, из текста в картинку и так далее. Такие алгоритмы обычно реализованы в ЭВМ.

О том, почему длина пароля важнее его сложности

(А. С. Лот)

В заметке «Что такое данные» говорится о том, что любые данные (сведения, информация) – это число. Пароль – это набор символов, являющийся набором какого-то множества заранее известных текстовых символов, обычно называемого набором символов или кодировкой. Пример простой кодировки – ASCII. Каждый символ кодировки в пароле вводится в поле, на котором установлен фокус курсора в текущий момент, с электронной (аппаратной) или экранной клавиатуры до тех пор, пока не будет достигнут предел длины текста в поле ввода либо пользователь не захочет отправить введенный текст пароля на сервер нажатием специально отведенной кнопки. Сложность пароля определяется мощностью множества его элементов, длина того же пароля – мощностью мультимножества входящих в него элементов (символов текстового набора). К сожалению, существующие компьютерные технологии не рассчитаны на правильную обработку пароля как единого объекта данных, представляющего собой одно число, а вместо этого алгоритмы обрабатывают пароли посимвольно, рассматривая его как последовательность чисел, где каждому символу назначен его числовой код, изначально известный и заданный кодировкой. Такие числа могут сравниваться попарно с соответствующими кодами образца пароля в проверяющей подпрограмме либо различными комбинациями арифметических операций над кодами символов сводиться к одному числу, которое тоже будет сравниваться с эталоном в проверяющем алгоритме, чтобы, к примеру, не хранить в нем сам пароль в текстовом отображении или ускорить сравнение за счет более эффективного использования пропускной способности сетевого канала связи и сокращения вычислительных затрат на перебор и попарное сравнение элементов. Для сверки введенного пароля с эталоном необходимо конечное значение времени центрального процессора, затрачиваемое на вычисления, производимые над числовыми данными, и конечная величина памяти, используемая для хранения введенного пароля, эталона и промежуточных результатов, и еще канал связи с конечной пропускной способностью, потому что эталон пароля чаще всего хранят в защищенном хранилище удаленного сервера в целях соблюдения стандартов безопасности. Для взлома аккаунта злоумышленник может подбирать пароли перебором входных последовательностей символов и, основываясь на времени отклика серверной машины на тот или иной вариант, выбирать более подходящие наборы символов, причем он может менять как длину пароля, так и его сложность (разнообразие символов), поэтому, выбрав один из вариантов защиты – длину или сложность, нужно воспользоваться и другим, дополнительно усилив

свой пароль. Чем длиннее пароль, отправленный злоумышленником, тем дольше он будет отправляться на сервер по линии связи и тем дольше взломщику нужно будет дожидаться ответа, чтобы отследить результат, но разнообразие символов на эту задержку не повлияет. Задача злоумышленника – найти пароль, который подойдет быстрее всего, то есть время отклика на который будет стремиться к максимуму, так как пароль дальше проходит по стадии проверок, – это единственный верный пароль аккаунта, и для взломщика естественно пытаться увеличить время отклика, задача пользователя – защитить свои данные, для чего он должен сохранить пароль (и желательно – логин) в тайне, что может быть достигнуто равномерным распределением задержек для всех возможных для системы паролей, но при этом это не совсем реальная картина для существующих систем, потому что пароли в ней могут храниться и допускаться ко вводу совершенно различные. Пароль пользователя должен иметь равномерное распределение символов при его максимальной длине. Равномерное распределение максимально близко к белому шуму, что соответствует практически естественной случайности попадания при переборе, что сгладит шероховатости графика задержек у злоумышленника в пространстве наиболее длинных паролей, так как они допускают наименьшую вариативность при стремлении к наиболее случайному паролю у пользователя, что благоприятно для него. Удаленный подбор пароля требует задействования вычислительных мощностей и памяти на обоих концах канала связи. Каждый символ одной и той же кодировки занимает в памяти один и тот же размер области памяти и обрабатывается центральным процессором по одному и тому же алгоритму для всех чисел такой разрядности или, точнее, размера выделенной области памяти под это число (кодировки различаются алгоритмами работы с ними на более высоком уровне – уровне операционных систем), поэтому длинный пароль будет требовать тем больше памяти для обработки и тем дольше обрабатываться процессором, чем он длиннее, то есть содержит больше любых символов, но для сложности пароля это неверно: разнообразие паролей одинаковой длины не влияет на длительность обработки и расход памяти, если говорить об обработке при пересылке в канале связи и сравнении пароля с эталоном на другом его конце. Последовательность из одного символа можно угадать, перебирая последовательно N кодов его кодировки, а последовательность из двух символов уже подчиняется законам комбинаторики, и количество размещений на две разные позиции двух из $N+N$ элементов уже может превосходить в разы сумму мощностей одинаковых кодировок двух символов $N+N$ (ситуация с еще большим числом символов еще сильнее увеличит количество вариантов), поэтому пароли следует выбирать подлиннее, даже пароль из одних единиц будет надежен при достаточно большой длине, но

она ему потребуется достаточно большая, чтобы не было дурацкого мата. Что касается психологии людей, то они бывают ленивы и глупы, но законы психологии на взломщиков тоже распространяются. Здесь речь не только о математике, а именно о том, что важнее длина пароля, чем его наполнение. Чтобы это было правдой в случае с использованием осмысленного набора слов в качестве пароля, нужно соблюсти максимальную длину пароля, выбирая фразы подлиннее, – в некоторых системах уже делают возможность ввода очень длинного пароля, поэтому чтение этой статьи окажется полезным для пользователей, а когда осуществляют перебор пароля по словарю, то в первую очередь будут подставлять наиболее вероятные пароли, причем очень длинные пароли сделают объем словаря таким, что хранить и передавать его, а также использовать с подключением по сети, будет сложно или экономически нецелесообразно, ведь защиту в информационных системах организовать проще, чем взлом. Если максимальную длину пароля системы нащупать не удастся, надо использовать максимально длинный и сложный пароль (в библиотеках языков программирования для его генерации можно использовать генератор случайных чисел с псевдоравномерным распределением), но приоритет отдавать длине, почему – объясняется выше. Если взломщик при подборе пароля его генерирует без словаря, то ему потребуется значительно больше времени процессора и электроэнергии на генерацию. Можно представить, что одно осмысленное слово на естественном языке в тексте пароля является одним символом, тогда возможных вариантов пароля будет тем больше, чем больше слов в пароле, – это объясняет востребованность длины пароля в случае использования естественного языка в нем. С другой стороны, на стороне сервера может в первой линии защиты использоваться сравнение по хэшу пароля, а при коллизии – сравнение с хранимым на сервере полноценным паролем, тогда в случае коллизии применимы предыдущие рассуждения, а при применении злоумышленником радужных таблиц, содержащих словарь соответствий «хэш – строка пароля», с высокой долей вероятности будут подобраны только короткие пароли, поскольку хэш для хранения паролей обычно используют в виде буквенно-цифрового кода, а перебор такого хэша почти равносителен перебору длинного пароля, так как пространство поиска хоть и сужается, но не настолько, насколько при хэше в виде целого числа, причем коллизии в современных алгоритмах хеширования маловероятны, а это говорит о том, что набор хэшей и набор паролей в базе данных сервера – это почти взаимно-однозначное соответствие, при котором перебор с обратной генерацией пароля по его хэшу породил бы необходимость иметь огромного объема словарь, что нереально.

Непрерывный цикл исследования и классификация компонент автороведческой экспертизы (А. С. Лот)



Автороведческая экспертиза главной целью имеет установление авторства каких-либо произведений (идентификацию) и основана на предположении о возможности проведения такой идентификации. Центральным понятием в такой экспертизе выступает исследование, следующее из потребностей, предпосылки его возникновения, и завершающееся возникновением благ, постепенно снова порождающих потребности, что и образует замкнутый непрерывный цикл исследования в автороведческой экспертизе. Исследованием косвенно управляет черный ящик – организация (здесь не имеется в виду коммерческое предприятие), куда входят все исследователи, а также менеджмент, время и прочие ресурсы. Потребности, наличие которых предвещает появление исследования, логически подразделяются на интуитивные, непредсказуемые с точки зрения опыта и строго выверенные необходимостью, то есть научные. Вытекающие из исследования блага могут быть действительными, имеющими конкретную материальную ценность, или идеальными, принадлежащими духовной сфере жизни. Высокий интерес для рассмотрения исследователем представляют части, образующие исследование: требо-

вания (завизированные документальные ограничения исследования, исходящие от организации в ответ на потребности и применяемые к инструментам), инструменты (средства, приспособления, сведения, маркеры стиля, предназначенные для обработки исследуемых материалов с учетом требований, следующих цели идентификации авторских работ) и материалы (авторские произведения, которые предлагается идентифицировать исследователям, авторы, окружение и условия, в которых создавалось и хранилось произведение, и другие элементы, способные быть объектом исследования автороведов). Подробнее классифицируем в таблице составляющие исследования при помощи антонимичных пар русского языка, каждая из которых будет разбивать весь рассматриваемый набор на две части, причем их равенство не предполагается, а разбиение применяется к частям автороведческой экспертизы вообще.

Классификатор – компонент исследования в автороведческой экспертизе.

Требования

Естественные

Заложенные изначально природой вещей.

Экспериментальные

Установленные опытным путём.

Изменчивые

Меняющиеся на протяжении исследования.

Важные

Непосредственно относящиеся к области исследования.

Истинные

Имеющие подтверждённое положительное влияние на результат исследования.

Широкие

Применяются ко всем инструментам в текущем исследовании.

Инструменты

Материальные

Осязаемые.

Ручные

Не задействуют автоматические средства.

Фактические

Устроены сообразно логике.

Общедоступные

Сложность применения прогнозируема и известна и применение возможно.

Одиночные

Используют один алгоритм или одну формулу для получения результата.

Обвинительные

Помогают обличить авторские заимствования.

Регулярные

Подходящие к подавляющему большинству произведений.

Абстрактные

Абстрагирующие от конкретного языка.

Синхронные

Могут вместе реализовать одну цель исследования *материалов*.

Искусственные

Синтетические

Эмпирические

Узнаваемые наугад.

Постоянные

Неизменные на протяжении исследования.

Незначительные

Не имеющие прямого отношения к области исследования.

Ложные

Не имеющие подтверждённого положительного влияния на результат исследования.

Узкие

Применяются к части инструментов текущего исследования.

Нематериальные

Не имеют осязаемой структуры.

Автоматические

Используют автоматические средства.

Бездоказательные

Не имеют под собой никакого логического обоснования.

Сложновоспроизводимые

Результаты применения воспроизводимы только в особых условиях на специальном оборудовании (например, секретные расчёты на суперкомпьютерах).

Составные

Собирает инструменты, способные быть независимыми в одном исследовании.

Защитные

Помогают уточнить авторский стиль автору.

Редкие

Присущие отдельным группам произведений.

Конкретные

Использующие непосредственно языковые средства.

Разрознённые

Не могут реализовать одну цель исследования одновременно.

Однопроходные

Могут быть применены за один проход обработки.

Параметрические

Принимают на вход дополнительные параметры.

Точные

Не допускают ошибок классификации (например, признаки двоичной логики или неизменная длина слова).

Полезные

С доказанной высокой эффективностью.

Зависимые

Основанные на результатах применения других инструментов (например, среднее арифметическое длин слов)

Аппаратные

Не имеют программируемого модуля.

Быстрые

При увеличении объёма исследуемого материала сложность применения инструмента растёт медленнее, чем объём материала, то есть сложность по памяти растёт быстрее, чем сложность по вычислениям.

Внутренние

При изменении одного и того же материала изменяются и инструмент, и его результаты.

Гармоничные

Обследуют весь материал.

Глубокие

Использующие особенности отдельных наименьших единиц материала (например, номера букв текста в алфавите языка).

Грубые

Точность не превышает погрешность.

Младшие

Могут быть основой других инструментов.

Отрицательные

Исключают авторство рассматриваемой совокупности авторов.

Возвратные

Обеспечивают результат за несколько циклов.

Неуправляемые

Единственный параметр - части материала (например, количество букв в слове).

Приближённые

Допускающие погрешность.

Слабые

С доказанной низкой эффективностью.

Самостоятельные

Не зависят от результатов применения других инструментов (например, подсчёт количества букв в слове).

Аппаратно-программные

Работают по специально составленной программе.

Медленные

Сложность по памяти растёт медленнее, чем сложность по вычислениям.

Внешние

При изменении одного и того же материала изменяются только результаты применения инструмента, а инструмент не изменяется.

Хаотичные

Обрабатывают только некоторые части материала.

Поверхностные

Не используют особенности отдельных наименьших единиц материала.

Мягкие

Точность больше погрешности.

Старшие

Могут быть основаны на других инструментах (например, суперпозиция функций).

Положительные

Утверждают авторство рассматриваемой совокупности авторов.

Сползающие

Не требующие перестановки элементов *чистого материала* относительно друг друга для своего применения.

Законные

Не требуют нарушения закона для своего применения.

Горячие

Не требуют предварительной подготовки *материала* для своего применения.

Конечные

Не требуют останова исследователем.

Рекуррентные

Используют рекурсию.

Коллективные

Требуют участия нескольких операторов для своего применения.

Централизованные

Все операторы действуют в одном месте.

Шифрованные

Требуют защиты от утечки данных.

Предварительные

Не позволяют окончательно установить автора.

Прогрессивные

Точность идентификации автора возрастает с ростом объёма исследуемого *материала*.

Объективные

Человек не участвует в принятии решения.

Убедительные

Точность применения *инструмента* более 50%.

Принципиальные

Ход исследования *инструментом* прозрачен (например, алгоритм или формула, полностью известные исследователям).

Главные

Применение *инструмента* занимает большую часть времени хода

Расползающие

Требующие перестановки элементов *чистого материала* относительно друг друга для своего применения.

Незаконные

Не могут быть применены без нарушения закона.

Холодные

Требуют предварительной подготовки *материала* для своего применения.

Бесконечные

Не имеют ограничений по продолжительности анализа, поэтому требуют останова исследователем.

Нерекуррентные

Не используют рекурсию.

Индивидуальные

Требуют не более одного оператора для применения.

Распределённые

Операторы действуют в разных удалённых местах.

Нешифрованные

Не требуют защиты от утечки данных для своего применения.

Окончательные

Могут быть использованы для окончательного установления авторства.

Регрессивные

Точность идентификации автора убывает с ростом объёма исследуемого *материала*.

Субъективные

Человек участвует в принятии решения.

Неубедительные

Точность применения *инструмента* не более 50%.

Беспринципные

Ход исследования *инструментом* наблюдаем исследователем частично (например, искусственные нейронные сети).

Второстепенные

Применение *инструмента* не занимает большую часть времени хода исследования.

исследования.

Прогнозируемые

Не менее 3-х исследований показали эти инструменты как убедительные.

Подвижные

Содержат двигающиеся при работе механические части.

Чёткие

Колебания точности инструмента не превышают 10% при применении на всех известных произведениях одного автора.

Материалы

Природные

Образовавшиеся без участия человека.

Органические

Относящиеся к живому

Реальные

Осязаемые.

Измеримые

Доступные для непосредственной фиксации средствами измерений.

Дискретные

Различимые.

Алфавитные

С конечным установленным алфавитом.

Языковые

Алфавитные, образующие стройную систему.

Изменчивые

В виде потока на временном континууме.

Стихийные

Сформированные случайно.

Обиходные

Имеющие широкое распространение.

Стратегические

Предполагаемые достаточными для установления авторства.

Активные

Целенаправленно содержат много элементов, предназначенных для облегчения установления авторства.

Смешанные

Непредсказуемые

Менее 3-х исследований показали эти инструменты как убедительные либо инструменты неубедительны или не применялись.

Неподвижные

Не содержат двигающихся при работе частей (например, рентген).

Нечёткие

При применении на всех известных произведениях одного автора точность инструмента может колебаться не менее, чем на 10%.

Культурные

Созданные под влиянием человечества.

Неорганические

Не относящиеся к живому.

Виртуальные

Неосозаемые.

Скрытые

Невозможные к непосредственной фиксации средствами измерений.

Непрерывные

Неразличимые по отдельности.

Неопределённые

Не имеющие конечного установленного алфавита.

Несвязные

Не образующие стройную систему.

Постоянные

Неизменно фиксированные на протяжении исследования.

Синтетические

Результат целенаправленного творчества.

Специальные

Не имеющие широкого распространения.

Тактические

Предполагаемые необходимыми для установления авторства.

Пассивные

Не содержат специально множества элементов, предназначенных для облегчения установления авторства.

Чистые

Не могут быть полностью проанализированы, так как содержат лишние включения.

Динамичные

Изменяются с течением времени исследования.

Долгие

Частотные характеристики признаков *материала* близки в доверительном интервале общим показателям *материала*.

Лёгкие

Объём лишних включений не превышает объём *чистого материала*.

Толстые

При изменении объёма *чистого материала* на наименьшую его единицу весь *материал* и его авторская принадлежность не изменяются.

Проходимые

Содержат повторные элементы.

Растянутые

Содержат, повторяющиеся элементы, следующие подряд друг за другом.

Религиозные

Имеющие религиозное назначение.

Осознанные

Исследователи понимают *материал* (например, если он записан на родном языке исследователей).

Сомнительные

Авторы *материала* неизвестны.

Мгновенные

Существует только один *материал* этой разновидности, относимый к этой совокупности авторов (например, скульптор изготовил всего один бюст за всю жизнь).

Тихие

Материалы не содержат явных заимствований.

Сходные

Все *материалы* исследования принадлежат одной группе

Могут быть полностью проанализированы и не содержат лишних включений (например, подсчёт длины слова в буквах).

Статичные

Не изменяются с течением времени исследования.

Короткие

Частотные характеристики признаков *материала* выходят за пределы доверительного интервала общих показателей *материала*.

Трудные

Объём лишних включений превышает объём *чистого материала*.

Тонкие

При изменении объёма *чистого материала* на наименьшую его единицу изменяются весь *материал* и его авторская принадлежность.

Непроходимые

Не содержат повторные элементы.

Сжатые

Не содержат повторяющиеся элементы, следующие подряд друг за другом.

Нерелигиозные

Не имеющие религиозного назначения.

Неосознанные

Ни один смысл *материала* не понят.

Несомненные

Авторская принадлежность *материала* заранее известна исследователям.

Постоянные

Существуют более одного *материала* этой разновидности, относимые к этой совокупности авторов.

Шумные

Среди исследуемых *материалов* хотя бы один *материал* содержит явные включения (точные части) другого.

Несходные

Материалы исследования классифицируются по-разному.

классификатора (классифицируются по этой таблице *материалов* одинаково).

Самородные

Анализируются на языке оригинала.

Словесные

Содержат хотя бы два слова на естественном языке каждый.

Избыточные

Существуют менее одного века.

Однородные

Произведение с одним автором.

Упорядоченные

Языковые, содержащие знаки препинания.

Транспортабельные

Могут быть перемещены в другое место для исследования.

Разбросанные

Во время исследования части *материала* находятся в разных удалённых друг от друга местах одновременно.

Допустимые

Материалы, по которым возможно провести установление авторства.

Правильные

Большая часть *материала* соответствует правилам языка.

Чужеродные

Переведённые в другую систему обозначений (например, с одного языка на другой).

Бессловесные

Содержат менее двух слов на естественном языке каждый.

Недостаточные

Существуют не менее одного века и, вероятно, утратили автора и содержат искажения.

Разнородные

Произведение с несколькими авторами.

Беспорядочные

Не содержат знаков препинания.

Нетранспортабельные

Не могут быть перемещены в другое место для исследования.

Собранные

Во время исследования все *материалы* находятся в одном месте одновременно.

Недопустимые

Материалы, по которым невозможно установить авторство.

Исключительные

Большая часть *материала* не соответствует правилам языка или материал несвязный.

Подводные камни языка программирования C# (А.С. Лот)

Автор работает с использованием языка C# с 2009 года. За это время накопились следующие замечания по языку:

1. Его невозможно полностью изучить;
2. Он не работает напрямую так, как написано в коде;
3. Многие его средства неудобны, громоздки, не нужны в большинстве случаев;
4. Некоторые его средства лучше вообще никогда не использовать, потому что они либо устарели, либо ведут к ошибкам;
5. У него нет подробного детального описания того, как всё устроено и работает;
6. В нём постоянно внедряется новый синтаксический сахар, который по сути бесполезен, но маркетинг призывает всех переписывать каждый раз старый сахар на новый;
7. Он не оптимален ни по памяти, ни по быстродействию;
8. Программы на нём тяжело отлаживать и тестировать, потому что используют в массе своей ООП;
9. Чтобы программировать на нём, нужно постоянно покупать новые книги и читать их;
10. Его фреймворки регулярно выкидывают на помойку, что опять ведёт к переписыванию всего и вся;
11. Большинство программистов на нём и слышать не желают про оптимизацию программ и культуру безопасного кодирования, как и про алгоритмы – обычно знания заканчиваются там, где заканчиваются руководства;
12. Его сложность и обилие классов заставляют задумываться над выбором средств реализации чаще, чем сосредотачиваться на алгоритме решения;
13. Для него мало бесплатных дополнений, большинство качественных сопутствующих средств – платны;
14. При постоянно растущей энтропии синтаксиса языка качественный код превращается в кашу;
15. Программисты, привыкшие к упрощениям, начинают бояться сложностей;
16. Обилие обёрток сводит на нет программирование на бумаге.
17. Абстрагированный от железа язык можно развивать до бесконечности.

18. Некоторые примитивы в операционных системах, отличных от Windows, могут вообще не работать вследствие особенностей реализации.

19. И ещё эта Visual Studio – один сплошной баг с регулярным обновлением

Как использовать популярную базу данных в качестве хранилища

1. Создать в расположении базы данных (БД) структуры хранения данных, принимающие от пользователей только двоичные данные;

2. Сформулировать порядок формирования оптимального для решения задачи запроса к БД в условно-защищённом расположении-приёмнике данных;

3. Получить данные от пользователя вашего продукта через интерфейс;

4. В приёмнике данных сжать данные и перевести их в двоичный формат;

5. Подставить двоичные данные в нужный подготовленный запрос и сжать его в двоичный формат, затем, согласовав формат передачи по условно-защищённым каналам, отправить запрос в расположение БД;

6. Дождавшись необходимого момента и условий, извлечь и запустить нужный запрос для сохранения данных в структурах, подготовленных на шаге 1.

P.S. Обзорщики данных должны располагать сведениями об использованном сжатии для извлечения данных, когда в этом будет необходимость.

Эффект тамбовской картошки в программировании

Современный бизнес связан с выполнением плана. За короткие промежутки времени обычно требуют от работников выполнить как можно больший объём работ или за продолжительный период времени хотят получить качественный продукт, т.е. бизнес хочет получить больше и лучше. Программисты, обслуживающие бизнес, часто хотят большего возмещения за свой труд, вложенный в производство продукта, или дополнительных опций за более качественные модули программ, т.е. работники хотят больше и лучше. Тем не менее, как правило, для бизнеса и специалистов больше и лучше означают разные вещи, т.е. они хотят не одного и того же: представитель бизнеса хочет быстрее доста-

вить продукт покупателю, вложить максимум удовлетворённых потребностей в каждый релиз и т.д., программист хочет избежать возможных багов и проблем в дальнейшей разработке по максимуму, самореализоваться через код, сделав его качественным и получив премию, и т.д. Несмотря ни на что, картошка у всех тамбовская. Тамбовщина издавна славилась своей картошкой, и сегодня, спрашивая о происхождении картошки, практически на любом рынке, в надежде получить качественный продукт к столу, мы обманываемся, получая ответ, что картошка тамбовская. То же касается выбора средств, методов и инструментов как в бизнесе, так и в изготовлении программного продукта: мы обманываемся, опираясь на опыт предыдущих поколений, не проводим самостоятельный анализ и выбор партнёров, стратегий и любых других вещей, позволяя бренду, внедряя маркетинг вместо мощных опор здравого смысла на переправе через бурлящий поток сомнений, подрывать будущее наших продуктов, перекладывая за собственные деньги и деньги предприятия ответственность на Васю, у которого в прошлые N лет был взрывной рост продаж, ведь покупали же другие, а мы чем хуже, когда есть деньги, драйв и воздушные замки, умело заранее подготовленные умным Васей. В результате между бизнесом и разработчиками разрастается бездонная пропасть, потому что у каждого лагеря формируются свои "науки" о должном образе действий, зачастую не имеющие никаких реальных под собой основ. Эти скрытые сложности порождают взаимное недоверие между бизнесом и специалистами, жить с которым придётся дальше, заваливая пропасть ещё большими деньгами, временем и людьми. Болезненный путь примирения бизнеса и программистов лежит через раскрытие поставщиков информации о тамбовской картошке и сведений о её продавце и истинном происхождении как со стороны бизнеса, так и со стороны работников.

Список использованной литературы

1. Лемей М. Agile для всех. – Питер, 2019.
2. Демин Д. E-mail-маркетинг. Как привлечь и удержать клиентов. – Питер, 2015.
3. Мартин Р. Чистый код. Создание, анализ и рефакторинг. – Питер, 2020.
4. Роббинс Д. Отладка приложений для Microsoft.NET. Мастер-класс. – Питер, 2008.
5. Маршалл Д., Бруно Д. Надёжный код. – BHV, 2014.
6. Маккарти Д., Маккарти М. Правила разработки программного обеспечения. – Питер, 2007.
7. Дыкан А., Севостьянов И. Увеличение продаж с SEO. – Питер, 2016.
8. Гласс Р. Факты и заблуждения профессионального программирования. – Символ-Плюс, 2007.
9. Сонмез Д. Путь программиста. Человек эпохи IT. – Питер, 2016.
10. Фаулер М., Бек К. Рефакторинг: улучшение существующего кода / М. Фаулер, К. Бек, Дж. Брант и др. – Символ-Плюс, 2019.
11. Тироу Ш. Видимость в Интернете. Поисковая оптимизация сайтов. – Символ-Плюс, 2009.
12. Макконнелл С. Совершенный код. Мастер-класс. – Русская редакция, 2019.

*Скачивайте и читайте онлайн остальные книги
на сайте автора: alexeylot.ru*

Содержание

Полезные высказывания из книги «Agile для всех» Лемея	5
Полезные высказывания из книги «E-mail маркетинг» Демина	9
Полезные высказывания из книги «Чистый код» Мартина	15
Полезные высказывания из книги «Отладка приложений для Microsoft .Net» Джона Роббинса	31
Полезные высказывания из книги «Надежный код» Бруно	35
Полезные высказывания из книги «Правила разработки программного обеспечения» Маккарти	46
Полезные высказывания из книги «Увеличение продаж с SEO» Дыкана	55
Полезные высказывания из книги «Факты и заблуждения профессионального программирования» Гласса	60
Полезные высказывания из книги «Путь программиста» Сонмеза ...	63
Полезные высказывания из книги «Рефакторинг. Улучшение существующего кода» Фаулера	64
Полезные высказывания из книги «Видимость в интернете» Тероу	70
Полезные высказывания из книги «Совершенный код» Макконнелла	80
Математизация автороведческой экспертизы (А. С. Лот)	122
Как выполнить код на Assembler из кода C# под Linux (А. С. Лот)	126
Синтаксический анализатор корректности текстовых арифметических выражений с использованием языка программирования Python (А. С. Лот)	128
Что такое данные (А. С. Лот)	129
О том, почему длина пароля важнее его сложности (А. С. Лот)	130
Непрерывный цикл исследования и классификация компонент автороведческой экспертизы (А. С. Лот)	133
Подводные камни языка программирования C# (А. С. Лот)	141
Как использовать популярную базу данных в качестве хранилища (А. С. Лот)	142
Эффект тамбовской картошки в программировании (А. С. Лот)	142
Список использованной литературы	144

