**Часть 1.О нашей профессии**

В связи с высоким ростом it-технологий, за последние 50 лет условия труда людей «нашей профессии» сильно изменились:

* Нет сильной зависимости от аппаратного обеспечения
* Процесс разработки полностью автоматизировался
* Расшились границы и возможности применения компьютеров

Одно остается неизменным. По-прежнему, программист – это человек, способный заставить компьютер решать поставленное перед ним множество задач.

* Программист = алгоритмизация и кодирование;
* Программист минус алгоритмизация = кодировщик
* Программист минус кодирование = постановщик задачи

На сегодняшний день, понятие «программист» - это очень широкое понятие. Однако, наиболее востребованным является специалист, умеющий самостоятельно формализовать задачу и решить ее.

! Не стоит путать определение тестера и профессии.

Иерархия подразделений в 80-е была более структурирована и намного строже, чем в наши дни. По ученой степени сотрудника можно было судить о его квалификации. Также, люди, имеющие должность техника («оператор ЭВМ») не допускались к проектированию. Этим занимались программисты-математики. В некоторых странах это разграничение до сих пор существует, однако у нас ситуация, к сожалению, несколько иная.

Софтостроение нельзя назвать искусством. Определение «техническое творчество» подходит больше. Это ни ремесло, ни искусство, ни наука, но, как говорится «aurea mediocritas», поэтому к делу стоит подходить творчески. Все в руках разработчика.

Программирование, как и все производство уходит в сферу услуг. И это логично!

Понятие «круговорот» в софтостроении можно определить как постоянная миграция разработчиков, которые вышли за штат и переквалифицировались в новых специалистов, в связи с устаревшей, той или иной технологией разработки.

При этом конкуренции практически не существует, т. к. заказчику комфортно работать со знакомым разработчиком. Замена подрядчика - очень рискованная процедура.



* Держитесь подальше от того места, где конкурентами будут 6 миллионов человек, чтобы жить спокойно.

Основные варианты для начинающего разработчика:

* найти проект с уже набившими шишек заказчиками и квалифицированными менеджерами, но места для поиска можно пересчитать по пальцам.
* специализация на предметных областях. В этом случае разработчик относительно автономен.

! Хорошо оплачиваемая работа с творческим подходом к труду в современном мире – это привилегия, за которую придётся бороться всю жизнь.

**Небольшой словарик для начинающих:**

1. «Быстро растущая компания» – фирма наконец получила заказ на нормальные деньги. Надо срочно нанять народ, чтобы попытаться вовремя сдать работу.

2. «Гибкие (agile) методики» – в конторе никто не разбирается в предметной области на системном уровне. Программистам придётся «гибко», с разворотами на 180 градусов, менять свой код по мере постепенного и страшного осознания того, какую, собственно, прикладную задачу они решают.

3. «Умение работать в команде» – в бригаде никто ни за что не отвечает, документация потеряна или отсутствует с самого начала. Чтобы понять, как выполнить свою задачу, требуются объяснения коллег, как интегрироваться с уже написанным ими кодом или поправить исходник, чтобы наконец прошла компиляция модуля, от которого зависит ваш код.

4. «Умение разбираться в чужом коде» – никто толком не знает, как это работает, поскольку написавший этот код сбежал, исчез или просто умер. «Умение работать в команде» не помогает, проектирование отсутствует, стандарты на кодирование, если они вообще есть, практически не выполняются. Документация датирована прошлым веком. Переписать код нельзя, потому что при наличии многих зависимостей в отсутствии системы функциональных тестов этот шаг мгновенно дестабилизирует систему.

5. «Гибкий график работы» – программировать придётся «отсюда и до обеда». А потом после обеда и до устранения всех блокирующих ошибок.

6. «Опыт работы с заказчиком» – заказчик точно не знает, чего хочет, а зачастую – неадекватен в общении. Но очень хочет заплатить по минимуму и по максимуму переложить риски на подрядчика.

7. «Отличное знание XYZ» – на собеседовании вам могут предложить тест по XYZ, где в куске спагетти-кода нужно найти ошибку или объяснить, что он делает. Это необходимо для проверки пункта 4. К собственно знанию XYZ-тест имеет очень далёкое отношение.

Тесты – особый пункт при найме. Чаще всего они касаются кодирования, то есть знания синтаксиса, семантики и «что делает эта функция».

**Основные принципы составления резюме:**

* Краткость – сестра таланта
* Кто ясно мыслит, тот ясно излагает
* Не фантазируйте
* Тем более не врите
* Упирайте на технологии
* Не делайте ошибок
* Будьте готовы, что далее первой страницы ваше резюме читать не станут

! Всегда существуют обходные пути

**С чем согласен:**

Практически во всем согласен с автором. Особо стоит отметить, что я также склонен к мнению о строгом разграничении между техником и проектировщиком.

Согласен с мнением, что софтостроение –это ни наука, ни искусство, ни ремесло, но, я бы добавил, что оно может быть одним из них в зависимости от разработчика.

Остальное носит больше ознакомительный характер.

Очень понравился словарик для начинающего.

**С чем не согласен:**

Не согласен с мнением, что нужно держатся подальше от мест, где конкурентами будут 6 миллионов человек. Мне кажется, что тут зависит от самого человека и каких целей он хочет достичь.

**Тема для эссе:**

Составление резюме

**Часть 2.**

Теперь, когда нужная информация всегда под рукой в большом объеме, необходимо уметь отыскать в этом объеме необходимое.

На данный момент мы не стоим на месте, а идем с большой скоростью все дальше и дальше. Трудно представить, как изменится наш мир в будущем. Однако, т к мир программных технологий основан на математических и лингвистических моделях и подчинён законам ведения бизнеса, можно пытаться делать прогнозы. Поскольку эти законы, на данный момент, строгие, то сильных изменений в самом софтостроении ожидать не следует.

Собирание «кубиков» выросло в большой рынок.

Для аппаратуры используется модель конечного автомата.

* она обеспечивает полноту тестирования.
* компонент работает с заданной тактовой частотой, то есть обеспечивает на выходе сигнал за определённый интервал времени.
* внешних характеристик (состояний) у микросхемы примерно два в степени количества «ножек», что на порядки меньше, чем у программных «кубиков».
* высокая степень стандартизации даёт возможность заменить компоненты одного производителя на другие, избежав сколько-нибудь значительных модификаций проекта.

Помимо модульного теста нужно программировать тест производительности, таким образом можно предположить, что устройство будет работать.

Собрав из компонент программу, ограничиваются выборочным тестированием, полагаясь на вероятность.

Программирование без исходников или черные ящики

Когда вы имеете дело с черным ящиком, вы до конца не понимаете, как он работает. В конечном итоге, это может привести к безысходности и безнадеге.

! Отсутствие системы управления версиями кода – один из случаев безысходного программирования.

Не стоит уподобляться новизне, когда ядро остается практически неизменным.

Иногда проверенные средства выигрывают у навороченных, при этом экономя время. Однако инновации обеспечивают преимущество, но порой даже простые задачи вызывают уйму времени.

Развитие ООП вытеснило женский труд из отрасли.

**С чем согласен**

Абсолютно поддерживаю автора по поводу безысходного программирования. Ведь слепое использование фреймворков и библиотек ведет к непониманию процесса работы программы и к уменьшению ее производительности. К сожалению, многие сейчас забывают об этом, ошибочно полагаясь на свою сообразительность. Однако я, в процессе разработки, всегда лезу «под капот», пытаясь понять, как эта штука работает. И всем советую!

Очень понравилось мнение о женщинах – программистах.

**С чем не согласен**

Автор намекает на то, что не стоит гнаться за новизной, когда платформа практически не меняется. Так то оно так, но, на мой взгляд, оптимальный подход – изучать новое, не забывая старого.

**Темы для эссе**

Роль женщины в программировании.

Фреймворки и библиотеки

**Часть 3.**

Смысл всей истории в том, что установленные стандарты при проектировании и возможности разработчика, опирающиеся на современные технологии, дают возможность процессу разработки бывает очень сложным и не всегда оптимальным. К сожалению, от этого страдают также и начинающие разработчики.

Начиная с 1994 получили развитие языки «под веб». Лежащий в основе названных платформ принцип был просто замечательным, хотя и совсем не новым. Логика приложений реализовывалась на стороне сервера скриптами на интерпретируемом языке, тонкий клиент-браузер в качестве терминала только отображал информацию и ограниченный набор элементов управления вроде кнопок.

Для веб-приложения:

1) реализация этого сценария одними серверными скриптами невозможна, необходимо задействовать клиентские.

2) необходимо хорошо представлять себе механизмы взаимодействия браузера и веб-сервера, чтобы синхронизировать вызовы и организовать передачу статуса.

3) веб-приложение не имеет состояния, поэтому понятие пользовательской сессии очень условное.

Помимо прикладной задачи, нужно решить кучу проблем. В основном, это платформенная независимость. В этой связи было найдено решение — браузер.

Получили развитие такие вещи как Google Сhrome и Internet Explorer.

! Следует аккуратно выбирать технологии для веб-интерфейса.

Прекращение распространения Java с Windows:

1) судебная тяжба Sun в 1997 году с Microsoft.

2) универсальность и кроссплатформенность среды, обернувшаяся низким быстродействием и невыразительными средствами отображения под вполне конкретной и основной для пользователя операционной системой Windows.

3) необходимость установки и обновления среды времени исполнения (Java runtime).

Так или иначе, необходимость в кросс-платформенных приложениях оставалась.

Поскольку для динамического содержания веб-сайтов использовались браузеры со скриптами, то разработка корпоративных приложений становилась непригодной. Первые попытки для решения проблемы предпринял Microsoft со своими WinForms, но очень скоро от них отказался и, как следствие, неудача.

Переход к WPF, а затем и Silverlight. Из-за проблем с ASCII-файлами и отсутствии прямого доступа к БД, на смену приходит HTML5 и множество фреймворков. Большинство «старых» технологий Microsoft замораживается, оставляя разработчиков в интересном положении.

**С чем согласен**

В большинстве своем данный фрагмент книги носил повествовательный характер. Автор рассказал о развитии веб-технологий так, как он видел это своими глазами, поэтому спорить с автором считаю бесполезным. Все очень интересно и увлекательно. С мнением автора трудно не согласится.

**С чем не согласен**

Почему-то вся информация подается со стороны компании Microsoft. Ничего не сказано о различных браузерах и развитии JavaScript.

**Темы для эссе**

Развитие браузеров

Упадок IBM

Как бродил Microsoft в веб-технологиях..

**Часть 4.**

* 1970 -1980 «С++ против Паскаля»

В следствие (в основном С++), широкая популяризации ООП.

Главный довод: ООП позволяет увеличить количество кода, которое может написать и сопровождать один среднестатистический программист. Все бы хорошо, но возникло такое понятие, как «ад паттернов».

Определение «ада паттернов»:

* слепое и зачастую вынужденное следование шаблонным решениям.
* глубокие иерархии наследования реализации, интерфейсов и вложения при отсутствии даже не очень глубокого анализа предметной области.
* Вынужденное использование все более сложный и многоуровневых конструкций.
* лоскутная 38 интеграция существующих систем и создание поверх них новых слоёв API.

В результате, программистам приходится писать свои методы согласно архитектурной специфике той или иной иерархии. Самые простые методы реализуются при помощи подключения многоуровневых конструкций. Разумеется, превратить код программы в тарелку спагетти можно без особого труда.

Поскольку язык С++ достаточно «тонкий» и человеческий фактор – решающий на проекте, появились более безопасные языки, похожие на С++: сначала Java, потом С#.

ООП очень требовательно к проектированию.

Изящнее выглядят интерфейсы. Но если в реальном мире книга, она и в музее – книга, то во вселенной интерфейсов «книга в музее» – неопознанный объект, пока не реализован соответствующий интерфейс «экспонат».

Современные платформы перегружены огромным количеством классов и методами в них. Однако, здесь, как и в обычной речи, большинство классов и методов не используется. Все это экзотика и намного рациональнее использовать не столь «увесистые» платформы для разработки.

ООП, несмотря на иерархические минусы, все же заставляет задумываться разработчиков об архитектуре. Перед ними стоит задача не просто составить базовый каркас приложения, но и сделать его гибким, чтобы в любой момент его можно было расширить.

В связи всем этим, на сегодня мы имеем следующую ситуацию: большинство проектов сегодня используют ООП. Это обеспечивает безопасность продукта, хотя и ставит преграды(ад паттернов, фреймворки). Некоторые принципиально отказываются от ООП.

Сокрытие базы данных

Вместе с миром ООП развитие получил так называемый реляционный мир. Появились СУБД:

1) удалось в 1980-х годах освободить программистов от знания ненужных деталей организации физического хранения данных.

2) большинство форматов данных, которыми оперируют программы, хорошо ложатся на модель двумерных таблиц.

Это и предопределило их успех.

В отличии от ООП, реляционные СУБД имели строгую математическую теорию в основании => были реализованы попытки сделать тоже самое и с ООП, однако ни к чему хорошему это не привело.

И реляционная и объектная модели относятся к логическому уровню проектирования программной системы т е можно создать одну и ту же систему используя ООП или реляционный подход.

Скрещивание «ежа с ужом»

Сложилась такая ситуация, что программы пишутся с использованием ООП, а данные хранятся в реляционных БД. В этой связи сформировался подход — ORM, реализующий этот переход.

Эволюция разработки проекта с ORM:

1) выбор ORM-фреймворка для отображения.

2) реализовывать модель предметной области. Добавляем классы, свойства, связи. Строим CRUD.

3) Собственно сама реализация.

Сравнение SQL и NHibernate.

В том же SQL Server для рекурсий и гонок ничего не сделано => система нестабильна.

**Общее мнение после прочтения**

Ну что ж, если при чтении предыдущей части возникали какие-то подозрения, то сейчас они полностью подтвердились. Автор явно настроен против всего современного мира разработки и технологий. Если говорить об ООП, то в целом в мнением автора можно согласится. «Ад паттернов» и все из него выходящее действительно имеет место. Однако, когда речь зашла про ORM, автор явно перегнул палку. Дело в том, что я в корне не согласен и не понимаю аргументов, которые были приведены.

Итак, по порядку. Почему использовать собственный язык ORM это плохо? Сама идея объединения ООП и БД очень интересна. Тем более, что есть очень неплохие примеры.

Теория и стандарты под ORM. И опять, автор явно навязывает, что с ними что-то не так.

Сравнивая SQL запрос и работу c NHibernate автор вообще промахнулся. Было сказано, что работать с Nhibernate трудно и неудобно, тк очень громоздко получается. Так случилось, что я сталкивался с этим и понимаю о чем идет речь.

Давайте взглянем на ситуацию со стороны Java. Мы имеем JDBS технологию работы с SQL запросами с одной стороны и Hibernate с другой. Понятно, что речь идет о проекте, и в лучшем случае вы сделаете около 50 запросов в БД. Использую JDBS вы каждый раз будете генерировать этот запрос. В итоге, вы получите полотно, а не структурированный код. Что получаем с Hibernate. Если вы грамотный разработчик, то вы попытаетесь реализовать своего рода каркас в виде абстрактного класса или интерфейса. Потом, при острой нужде, вы его просто расширите или переопределите нужные методы. В итоге, те же самые методы могут использоваться для разных объектов, код понятен и структурирован. Все логично и понятно. Да, в Hibernate можно использовать NativeQuery, что по сути является аналогом JDBS.

Теперь по поводу ситуации с гонкой SQL запросов. Да, возможно это не предусмотрено и это прокол. Но я не могу представить ситуацию, когда бы запросы в БД могли бы находится в состоянии гонки, поскольку выполняются они очень быстро в одной транзакции.

Ну и последняя капля — это высмеивание методологии скрам и, как следствие, вывод о нестабильной системе.

Конечно, мнения автора интересны и, скорее всего, описанные им проблемы имеют место. Однако, на мой взгляд, если профессионально подходить к разработке, этих проблем можно избежать.

**Темы для эссе**

1) ORM фреймворки

2) безопасность запросов в БД

**Часть 5.**

Мнение

Конкретно в этой части книги мне трудно высказать свое мнение, поскольку автор повествует, в основном, о развитии каналов связи, протоколов и сервисно-ориентированных архитектур. Интересно было узнать о том, как это происходило в СССР. Оказалось, что внедрение подобных систем было очень перспективно из-за централизации производства и административного аппарата. Как мы знаем, после прочтения все это не получило должного распространения.

Сам процесс работы с ВЦКП, по мнению автора, аналогичен работе с облачными вычислениями. Опять же, судя по тому, как это понятие описано в книге, сравнение очень удачное. Да и в целом, когда речь идет о новых технологиях или понятиях, автор всегда проводит аналогию со знакомыми вещами, акцентируя внимание на том, что иногда люди изобретают заново то, что уже есть на самом деле. Оно трудно для понимания и, как следствие, подвергается критике. Затем для конкретной проблемы появляется неизвестно кем разработанная реализация, и теперь все намного проще. Достаточно свести свою проблему к существующей, и решение уже есть. Однако, когда происходит переход к следующей стадии разработки выплывают на поверхность серьезным последствия этого выбора.

Вот, на мой взгляд, главная мысль автора. Возможно, это то, ради чего он написал эту книгу. Он приводил подобные доводы раньше, и сейчас опять он обращает на это внимание.

Однако, вернемся к насущному. Читая дальше, мы узнаем про технологию CORBA. В процессе своего развития и интеграции с различными платформами данная технология выливается в очень достойный инструмент для разработки. Однако из-за некоторых особенностей и проблем получили распространение другие сервисно-ориентированные архитектуры, которые якобы проще и знакомы нам(мне в частности). Но, обратив внимание на различные декларации и нелегкий путь бекенд разработчика в среде Java, трудно не согласится, что CORBA выглядит намного привлекательнее.

Возможно, автор пытается сказать, что мы движемся не в нужном русле. Я, к большому сожалению, не обладаю большим опытом, чтобы как-то комментировать это мнение, но аргументы автора мне показались очень уместными.

Прочитав небольшой отрывок про NET фреймворк, я опять сталкиваюсь со своей некомпетентностью. Я не вижу пока всех этих проблем. Как видно, в условиях современного рынка разработчику может быть не так комфортно. А с другой стороны, на то вы и разработчик, чтобы постоянно выходить из зоны своего комфорта и искать золотую середину между рискованными новыми и отодвинутыми зрелыми технологиями и концепциями.

Ах да, вот еще что. Еще одна вещь, которая мне очень понравилась — манипуляция терминами и производство услуг. Да, эта тема очень глубокая. И здесь я согласен с автором. Вообще, с некоторых пор, до меня стало доходить насколько важна роль маркетолога и насколько она отвратительна. Ведь эти люди манипулируя терминами часто заводят конечного пользователя в заблуждение. И как же смотреть на это все с открытыми глазами? Тут мне сразу вспомнился преподаватель, который всегда проповедовал «не верить никому, даже себе» или «черный — это не цвет, а название цвета». А мы так редко его слушали...

Темы для эссе:

1) CORBA. Подробнее

2) Децентрализация ресурсов. Курс взят.

3) Наша служба и опасна и трудна. Как чувствует себя разработчик сегодня?

**Часть 6.**

Разбирать в слоитости обязан каждый. С этих слов начинается данный раздел. Почему? Ответ тоже сразу можно найти. После этого, автор погружает нас в терминологию и объясняет, как устроен мир. Лично для меня, этот раздел был труден для прочтения. Автор делал теоретические выкладки, и приходилось перечитывать, чтобы понять смысл. Коротенько мы узнаем о логическом и физическом устройствах, о разных типах клиентов, даже немного о недостатках и преимуществах браузеров. Прочитав про уровни и слои, однозначно можно говорить, что очень много уже знакомо (когда-то слышал). Значит все это дошло и до наших дней и, в той или иной степени, используется.

Интереснее было читать о многозвенной архитектуре. Автор говорит о том, что каждая архитектура индивидуальна, а значит выбирать ее нужно с пониманием того, что делаешь. К сожалею, сейчас часто этим пренебрегают, беря в основу успешные примеры. Ну и как следствие большая часть функционала занимается перегонкой данных из одного формата в другой. Автор советует нам избегать подобного. Чем меньше прослойка между источником и GUI, тем лучше.

На этом заканчивается пролог этой части. Автор теоретически подготовил нас к основной истории, которая есть в этой главе, не оставив ни единого шанса усомниться в своей правоте. И действительно, каждое свое слово автор подтверждает случаем из жизни. Поэтому со многим я согласен, а если и есть вопросы, то это только из-за собственной неосведомленности.

Оставшаяся часть — это история, рассказанная главными ее героями. Разработчики рассказывают о периоде в их жизни с 1991 по 1994 год. В это время многие системно мыслящие люди начинали собственный бизнес из-за перелома начала 90-х.

За целой кучей непонятных для меня терминов очень трудно представить полную картину происходящего. Однако главную мысль словить можно. Все дело в том, что ребятам удалось создать весьма неплохую систему. В начале своего пути они не до конца осознавали весь смысл. Тем не менее, путем проб и ошибок, им удалось сначала написать Seller 1.0, а затем, из-за ограниченности этой системы, в кратчайшие сроки был написан фреймворк, который плавно перерос в язык NDL. Поразительно было то, что все это сделали по сути 3 человека менее чем за год. Полученная система Novell имела перспективы, но Windows 95 разбил их. На этом все и закончилось.

В качестве постскриптума выступает призыв о том, что нужно пробовать придумывать и реализовать свои идеи. История выше — явное тому подтверждение.

Мне история очень понравилась. Я очень рад за этих людей. Сейчас попытаюсь объяснить почему. Дело в том, что сам процесс создания вашего детища очень увлекателен и по-своему интересен. Но ничто не сравнится с тем моментом, когда ты заканчиваешь долгую работу над этим и видишь результат своего упорного труда. Это действительно волшебный момент, особенно когда результат очень хорош. Размышляя вот так, можно прикинуть с какими чувствами эти люди рассказывали нам эту историю. Что касается меня, то я хотел бы, чтобы у меня тоже была подобная история, которую я смог бы рассказать.

**Часть 7.**

Существует два основных подхода к разработке КИС, условно называемых «от производства» и «от бухгалтерии». Под термином «бухгалтерия» имеется в виду прежде всего внутренний, управленческий учёт на предприятии, а не фискальная её часть. Вот собственно и всё, что можно сказать про первую часть, которая фактически являлась продолжением той замечательной истории. Также стоит обратить внимание на усовершенствованную архитектру системы Ultima-S. Автор очень подробно ее описал и первое, что приходит в голову - «смотрите, как нужно делать». Дальше идет перечисление всех достоинств этой штуковины. Что ж, аргументы впечатляют.

Интереснее читать раздел «Нешаблонное мышление». Здесь мы снова сталкиваемся с проблемой, которую обсуждали уже много раз. Рациональное использование паттернов, как видно, является острой темой и для серьезных, повидавших жизнь, разработчиков, а не только для таких, как я. Если коротко, то автор говорит, что с шаблонами нужно считаться как ни крути. Кроме того, он отмечает, что найти достойную литературу/документации очень трудно, намекая, что порой люди не знают о чем пишут.

В связи с этим всем, логично сделать заключение, что подобные труды вредно читать тем, кто только начинает свой путь, т е мне. «Лучше изобретать свои велосипеды, чем пользоваться чужим шаблоном» - говорит нам автор. Может оно и так, но ведь ужасно интересно покопаться во всем этом и дойти до истины. Разве нет?

Самое забавное, что в конце концов автор советует использовать паттерны, а еще желательно думать при этом, что делаешь. Другими словами, нельзя мыслить шаблонно. Однако пару глав назад у него была принципиально другая позиция. Ну да ладно, возможно, мнение поменялось.

И последнее, сборка мусора. Идея в том, что встроенные сборщики зачастую имеют ряд недостатков, но иногда и преимуществ. Как пишет автор, черно-белых оценок здесь нет. Однако бывает, что случаются серьезные проблемы. Пример можно посмотреть в книге. Здесь я согласен с автором. Я считаю, что автоматизация такого рода задач — не лучшая идея. Ведь чем меньше этой автоматизации, тем больше свободы у разработчика. Я думаю, что на самом деле это так. Излишнее упрощение только мешает и снижает квалификацию разработчика. Вот взять Java, например. Еще в курсе лекций было сказано, что работу сборщика предугадать трудно. Лишь командой finalize() можно порекомендовать сделать чистку. Но это не спасает вас, если вы не следите за тем, что делаете. Существует очень много способов получить утечку памяти. (подробнее тут: <http://habrahabr.ru/post/132500/>). Я помню, у одно моего одногруппника были с этим проблемы, когда на консоль вылетело OutOfMemory. В С++ работать в этом плане было приятнее.

Тема для эссе: типы OutOfMemory в Java и как их ибежать.

**Часть 8.**

**Журнал хозяйственных операций**

Рассмотрим конкретные примеры разработки учетных приложений (антипаттерн «Таблица остатков»):

1. Нельзя отделять количество товара на текущий момент от действий, в результате которых было получено это значение.

Действительно, в случае ошибки, велик риск, что и дальше программа будет считать неверно. Нельзя доверять пользователю.

1. Захват данных критической секцией(транзакцией) тоже не решает проблему.

Такой подход обязательно приведет к очереди за доступом к данным. Чтобы совершить покупку, необходимо будет ждать, пока будет куплен предыдущий товар.

1. Хранить историю остатков нужно.

Чтобы найти значение остатка необходимо обработать все произведенные вычисления неслабым функционалом. При этом скорость вашей программы уменьшается в разы.

1. Нужно хранить остаток за определенный период и перечитывать его при изменении данных –> триггер.

На мой взгляд, триггеры – не самый лучший вариант в любом случае. Использование их часто приводит к непониманию работы всей системы. Особенно когда над программой усердно трудятся несколько разработчиков. Однако, в данной системе, похоже, что триггер – единственное решение проблемы.

**UML и птолемеевские системы**

UML:

• не универсальный, а унифицированный, объединяющий отдельные практики;

• не язык, а набор нотаций (графических);

• не моделирования, а в основном рисования иллюстраций, поясняющих текст многочисленных комментариев.

Две основные альтернативы:

* использовать как есть
* использовать как моделирование и генерацию кода

Очень понравилось сравнение с птолемеевской системой. Понятие UML сразу стало более прозрачным. Ведь на самом деле идея UML сама по себе весьма интересна. Но снова автор предупреждает, что ко всему интересному стоит подходить с осторожностью. В случае с UML, его неправильное использование в самом начале (вспоминаем Киплинга) повлекло за собой соответствующую цепочку. Мало кому удается использовать UML с высокой отдачей. Да и непонятно, как эффективно использовать этот язык. Хотя и языком эта штуковина не является. Как видите, все здесь очень запутанно. И, конечно же, автор приводит целый ряд недостатков. В частности, когда работа двух специалистов над одной системой дала не сопоставимый результат.

Однако у всего этого есть и достоинства. Использование UML позволяет хоть как-то собрать все запросы заказчика и обрывочные сведения, затем формализовать и конкретизировать. В результате какой-нибудь результат точно будет.

Мне непонятно, почему не предпринимается попыток улучшить UML или создать что-либо принципиально новое с теми же целями. Ведь идея создания унифицированного языка очень перспективна и интересна.

**Часть 9**

Мнение

В очередной раз мы возвращаемся к проблемам разработки в связи с быстротечным развитием технологий. Дело в том, что рядовой разработчик, потратив годы на детальное изучение конкретной технологии, через некий промежуток времени осознает, что технология утрачивает свою актуальность. Чтобы сохранить должность, он пытается доказать обратное. И, таким образом, никакого движения вперед нет. К сожалению, такое явление, на мой взгляд, свойственно не только разработчикам, но и всему современному обществу.

Читая этот отрывок, я все ждал, когда автор предложит какое-нибудь решение этой проблемы. Впереди ждало разочарование. Какого-то решение нам автор не предложил. Вместо этого, нам была предложена история о том, как ребята боролись с этим зверем и, спотыкаясь, создавали современный мир технологий. После этого, автор делает заключение, что квалификация специалистов падает и быстрое развитие технологий одна из причин. Особенно это касается специалистов в области баз данных.

На мой взгляд, самая интересная часть этой главы это та, где автор рассказывает как решалась проблема использования средств для хранения информации у пользователей с ограниченными правами. Да и как вообще разворачивать большое приложение на стороннем компьютере, где нет необходимых инструментов.

Тема для эссе: Как донести технологию до конечного пользователя?

**Часть 10.**

Конспект

Работа группы программистов без системы контроля версий годами ведет к возникновению «цифровой пыли десятилетия». И, как следствие, появляются антипрактики.

Пример: ERP-система переходящая от файловой системы к СУБД и образование при этом таблицы-справочника.

Следовательно, ревизия кода очень полезна:

* эта процедура регулярная и запускается с момента написания самых первых тысяч строк;
* процедуру проводят специалисты, имеющие представление о системе в целом.

«Уровень QA-специалиста должен быть не ниже архитектора» Дж. Фокс

Хороший способ проверки кода на копирование:

Создавать zip-архив файлов кода, если размер растет медленнее, чем размер самых файлов => код постоянно копируется, а не форматируется.

«Приходишь в отечественную компанию, смотришь, как у нее устроено IT, и видишь, что люди просто упали с дуба.» М. Донской

Статистика Кэпера Джонса:

• среди проектов с объёмом кода от 1 до 10 миллиона строк только 13 % завершаются в срок, а около 60 % свёртываются без результата;

• в проектах от 100 тысяч до 1 миллиона строк эти показатели выглядят лучше (примерно 25 % и 45 %), но признать их удовлетворительными никак нельзя;

• в проектах примерно от 100 тысяч строк на кодирование уходит около 20 % всего времени, и эта доля снижается с ростом сложности, тогда как обнаружение и исправление ошибок требует от 35 % времени с тенденцией к увеличению.

В любом софтостроительном процессе, будь то заказной проект или продукт для рынка, всегда можно выделить 4 основные стадии:

• анализ, чтобы понять «что делать»;

• проектирование, чтобы определить и запланировать «как делать»;

• разработка, чтобы собственно сделать;

• стабилизация, чтобы зафиксировать результат предыдущих этапов.

Заказчик, осознавая свою несхожесть с мифологическим титаном мысли, он может достаточно быстро увидеть сформулированные требования и сценарии в реализации, отлитыми, разумеется, не в бетоне, а в гипсе, и на практике понять их противоречивость и неполноту.

Синтез «водопада» сложной системы, итоги проектирования которого подаются на вход «гибкой» производственной машины кодирования и стабилизации – что может быть бессмысленнее и беспощаднее?

**Модульные тесты** тоже бывают сложными, а значит, с высокой вероятностью могут содержать ошибки. Тогда возникает дилемма: оставить всё как есть или перейти к мета-тестированию, то есть создавать тест для теста.

Мнение

Мне кажется, что эта часть конспекта близка больше опытным разработчикам. Все дело в том, что теперь в книге все глубже рассматриваются мелкие детали самого процесса разработки. Все это безумно интересно, но есть и другая сторона. Заключается она в том, что своего мнения нет, а для того чтобы его заиметь нужно несколько лет поучаствовать в реальной разработке. Сейчас, когда нужно выразить свое мнение, немного теряешься, поскольку повествование становится все глубже.

Сначала хочется сказать о книге в целом. Очень приятно, что она изобилует мелкими советами, которые могут очень помочь в разработке. Вот и здесь можно найти отличный совет для эффективного рефакторинга кода. Примерно в том же месте, можно найти описание самой проблемы в целом и, конечно же, пример из жизни по данной теме. К сожалению, далеко не все главы книги имеют такую структуру, однако заметно, что автор старается держаться выбранного направления.

Теперь об основной теме отрывка, о гибкости. Честно говоря, приятно было обнаружить, что большая часть посвящена методологиям, которые мы очень подробно рассмотрели месяц назад. В моем понимании, гибкая система – это такая система, которая быстро адаптируется, изменяется и расширяется в соответствиями новыми требованиями от заказчика. Я думал, что построение такой системы целиком и полностью лежит на разработчике. На самом деле, спроектировать, а тем более реализовать такой подход очень сложно, да и задаче это очень широкая. Автор даже рассмотрел пример гибкой методологии и выделил основу – работа в команде и ее быстрое взаимодействие и взаимопонимание. Но даже при таких масштабах создание гибкой системы очень неоднозначная и сложная задача, и часто приходится приходиться решать такие проблемы как зацикливание.

**Часть 11.**

Конспект

**Team Foundation Server** (сокр. TFS) — продукт корпорации [Microsoft](https://ru.wikipedia.org/wiki/Microsoft" \o "Microsoft), представляющий собой комплексное решение, объединяющее в себе [систему управления версиями](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D1%83%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F_%D0%B2%D0%B5%D1%80%D1%81%D0%B8%D1%8F%D0%BC%D0%B8), сбор данных, построение [отчётов](https://ru.wikipedia.org/wiki/%D0%94%D0%BE%D0%BA%D0%BB%D0%B0%D0%B4), отслеживание статусов и изменений по проекту и предназначенное для совместной работы над проектами по [разработке программного обеспечения](https://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F). Данный продукт доступен как в виде отдельного приложения, так и в виде серверной платформы для [Visual Studio Team System](https://ru.wikipedia.org/wiki/Visual_Studio_Team_System" \o "Visual Studio Team System) (VSTS).

Работа с “джинном”:

Модель в виде XML-файлов поступает на вход «заклинателю». Производятся проверки непротиворечивости модели, выдающие ошибки либо предупреждения разной степени важности. Во время анализа модель также преобразуется во внутренний формат в

виде множества объектов с открытыми интерфейсами доступа. Если модель корректна, «заклинатель» начинает призывать «джиннов» сделать свою работу, передавая каждому на вход кроме самой модели ещё и разнообразные параметры, конфигурацию. Обработав модель в соответствии с конфигурацией проекта, джинн выдаёт готовый к компиляции в среде разработки код. Для слоя хранения данных кроме генерации специфичных для СУБД SQL-скриптов производится их прогон на заданном сервере разработки. В случаях, когда система уже существует и подлежит, например, переделке, можно восстановить модель из схемы базы данных. Проведя один раз импорт, далее мы редактируем, структурируем модели и продолжаем работать только в обычном цикле изменений «через модель».

Слои:

1. Слой хранения (СУБД)

2. Слой домена (Nhibernate)

3. Слой веб-служб и интерфейсов доступа (ServiceStack)

Мнение

Эта часть началась очередной увлекательной историей. На этот раз под критику автора попал TFS, а именно Team Foundation Server, а именно трудности в его установке, настройке и эксплуатации. Здорово, что автор предоставил, по сути, подробную инструкцию по настройке TFS. По структуре (да и по содержанию) глава напомнила мне дневник доктора Борменталя из замечатьльной повести Булгакова “Собачье сердце”. Всей команде разработчиков и аддминистраторов вместе с автором изрядно пришлось попотеть с этой штуковиной. Автору даже не хочется вспоминать об этом, поэтому уже традиционного итога и вывода в этой главе нет.

К большому сожалению, такая ситуация, как мне кажется, не редка. Практически всегда огромное количество времени уходит на установку средств для разработки, их конфигурацию и изучение документации. Более того, это один из самых сложных и мучительных этапов разработки, поскольку после того, как все настроено, вам остается наслаждаться написанием вашего кода. Этот процесс, в идеале, хорошо отлажен и приносит море положительных эмоций. Но вернемся к проблеме. В книжном случае, весь этот тяжелый процесс был украшен большими требованиями для установки и не вполне понятной работой самой системы (даже консультантам и экспертам). Почему так происходит? Наверное, дело в масштабах этого инструмента разработки и объеме ресурсов, требуемых для его работы. Чем выше эти показатели, тем больше зависимостей, которые нужно каким-то образом учесть при настройке. Тот, кто предоставляет эти средства для разработки обязан редусмотреть эти вещи, но в нашей истории явно не тот случай.

Идея генерации кода с помощью “заклинания” мне очень понравилась. Это удобно. Передав на вход модель с необходимыми параметрами, получаем сгенерированный код. Такой подход увеличивает скорость разработки в разы. Но с другой стороны, мы возвращаемся к той же насущной проблеме. Дело в том, что мы не знаем, как сработает “заклинание” и даст ли оно то, что нам надо. Об этом мы говорили, разбирая фреймворки и ад зависимотей.

Чтобы как-то осознать весь принцип работы джина в лампе последняя часть этой главы – описание сущностей системы и слои хранения. Все можно увидеть наглядно, а при желании и написать нечто похожее. Здорово.

**Часть 12.**

Конспект

В жизни каждого мало-мальски сложного программного продукта есть стадия, когда система проходит некий порог увеличения сложности, за которым наступает состояние, которое я называю «самостоятельной жизнью».

Способы продвижения:

1. прекратить текущую разработку программы и начать новую, используя опыт предыдущего прототипа. На это кроме обычного мужества признания ошибок и риска полететь с должности начальника требуется ещё и немало денег.
2. выдать желаемое за действительное, побольше маркетинга, шуму, «подцепить» несколько заказчиков и на их деньги попытаться всё-таки перейти к первому. Такой трюк может сработать, если заказчику честно предлагают за какие-то вкусные для него коврижки побыть немного «подопытным кроликом», на котором система вскоре «должна заработать».

Если фирма с 50 % национального рынка электронных платежей вполне может работать без системы управления версиями исходного кода и дублированием таблиц в базе данных, то что же тогда говорить о стартапе…

«Вряд ли следует бояться заразиться насморком во время эпидемии чумы» - афоризм.

Современное софтостроение заслуженно забыто наукой. Ну а что вы хотите, если на вопрос «Почему нет науки на конференциях и в публикациях?» получаешь однозначный ответ «Никакая наука рынку не нужна». В результате на первый план выходят повара, написавшие очередную порцию рецептов приготовления пиццы и винегрета из найденных в холодильнике продуктов корпораций.

Нынешняя «гуглизация» позволяет быстро находить недостающие фрагментарные знания, зачастую забываемые уже на следующий день, если не час.

Мнение

В самом начале автор рассказывает о своем опыте работы в Париже, а именно в какие условия он там попал. То, что он там увидел не очень его обрадовало («При виде некоторых фрагментов кроме чисто русских эмоциональных выражений из меня периодически вылезало французское «что за бордель», вызвавшее похвалу моего коллеги, отмечавшего, что я делаю успехи в освоении языка.»). И вот, говоря о стадии самостоятельности проекта, автор затронул тему организации работы программистов. «Поскольку работу программиста трудно формализовать, случаются такие вот неприятности», - говорит нам автор. Может оно и так. Мне кажется, что сам процесс организации работы меньше всего завит от разработчиков. Все зависит от лидера в большей степени, от человека, который организовывает эту работу. Если же лидер попадается выдающийся, то, возможно, и формализовывать ничего не придется, и ситуаций с «самостоятельной жизнью» не будет. Сама организация совместной работы – очень трудная задача. Очень часто этим пренебрегают, хотя большая часть дальнейшего успеха зависит именно от этого.

Очень интересная ситуация получается. Сначала автор дал совет: если программист говорит, что это должно работать, то оно не работает, а если он говорит, что у него работало, то его нужно увольнять. Забавно, но компания, в которой автору посчастливилось поработать поступала именно таким образом со своим проектом. Теперь понятно, что автор имел в виду)

Дальше идет описание этой самой компании. Честно говоря, опытный разработчик воспримет это как обычное повествование. Ну а нам этот материал очень даже полезен. Мне кажется, что автор хочет передать досконально всю атмосферу разработки.

Вот «Три дня в IBM» очень интересная глава. Мне понравился 3-х страничный рассказ о долгом пути домой в Париж и проблему с платежными автоматами во Франции. Непонятно почему, но ночью платежные сервера во Франции «умирают». Да, программист – это болезнь. Ну а кроме этого, автор рассказывал, как он пытался решить проблему с кроссплатформенностью. Из всей главы понятно только то, что

1. Дядьки с IBM – это суровые пятидесятилетние волшебники, которые могут творить чудеса.
2. Даже опытные разработчики пробуют «на удачу» отладить программу.

Дальше мы слегка уходим в философию. К сожалению, трудно не согласится, что «никакая наука сейчас не нужна», и, с популяризацией «дот-комов», софтостроение умирает. И очень жаль, как по мне. Мы уже обсуждали это раньше. На данный момент очень легко попасть в «конвейер» и работать в нем лет 30, в котором о софтостроении полностью нет ни у кого представления. Гораздо сложнее попасть в команду, где непосредственно решаются проблемы, разрабатывается что-то новое. Здесь можно развернуться на всю катушку и продемонстрировать всю силу математики, удивляя тем самым своих коллег, которые изучали математику не так глубоко, как мы. Да и свободы мысли здесь, как мне кажется, больше.

В следующей части автор дает советы по изучению литературы. Весьма полезная глава. Что ж, возьмем на заметку.

Очень оригинально написано о том, как все было создано. Тут нужно просто прочитать.

Ну вот, книга прочитана. Хотелось бы подвести итог из всего этого. После прочтения у меня осталось ощущения, что я лично беседовал в автором. Он выражал свои мысли в книге, а я в своих конспектах. Местами мнение автора было не совсем понятным, и я пытался изложить свое понимание (возможно ошибочное, но мы ведь учимся на ошибках). Ну а в целом заложен действительно огромный фундамент, многие вещи, изложенные здесь, теперь более прозрачны. Однако многое не понято до конца. Однозначно можно сказать, что интересно будет вернуться к книге и к своим конспектам, спустя несколько лет, набравшись опыта. Тогда окончательное мнение точно сформируется.

Темы для эссе:

1. Зависимость опыта от возраста