МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет прикладной математики и информатики
Кафедра технологий программирования

Дефрагментация мозга

Минск 2015

Оглавление

Глава 1. Конспекты ... 4
Белый Антон ... 4
Борисевич Павел ... 9
Гетьман Святослав ... 12
Григорьев Антон ... 36
Грушевский Андрей ... 53
Ипатов Алексей ... 62
Лебедев Николай ... 91
Михальцова Анна ... 106
Ровдо Дарья ... 113
Трубач Геннадий ... 145
Щавровский Святослав ... 157
Ярошевич Яна ... 165
Глава 2. Темы для эссе ... 188
Белый Антон ... 188
Борисевич Павел ... 189
Гетьман Святослав ... 190
Григорьев Антон ... 191
Ипатов Алексей ... 192
Лебедев Николай ... 193
Михальцова Анна ... 194
Ровдо Дарья ... 195
Трубач Геннадий ... 196
Щавровский Святослав ... 197
Ярошевич Яна ... 198
Глава 3. Отчеты ... 200
Отчет 1 (Щавровский Святослав) ... 200
Отчет 2 (Гетьман Святослав) ... 202
Отчет 3 (Лебедев Николай) ... 207
Отчет 4 (Щавровский Святослав) ... 210
Отчет 5 (Гетьман Святослав) ... 213
Отчет 6 (Лебедев Николай) ... 221
Отчет 7 (Щавровский Святослав) ... 223
Отчет 8 (Гетьман Святослав) ... 226
Отчет 9 (Лебедев Николай) ... 229
Отчет 10 (Щавровский Святослав) ... 231

Отчет 11 (Гетьман Святослав) 234
Отчет 12 (Лебедев Николай) 239

Глава 1. Конспекты

Белый Антон

В целом в главе ведется рассуждение автора о жизни программистов, что было, что стало. В частности, раздел «Специализация» сообщает важный момент, что сейчас ценятся больше люди, способные сами формализовать задачу, а также запрограммировать свое решение. Интересная мысль была в разделе «О красоте», что программирование сродни «техническому творчеству». То есть написанный вами код будет не только работать, но будет максимально понятен, компактен, аккуратен и вообще приносить удовольствие от прочтения.

К сожалению, необходимость быть экспертом в какой-либо области постепенно отпадает, ведь доля критичных к качеству проектов все меньше, да и любой, даже малообразованный, человек при желании может заучить полезный материал и вставлять его везде, где можно: в резюме, при собеседовании, за обедом. И при этом его с большей вероятностью возьмут на работу, нежели вас, ведь ему можно в разы меньше платить. Таким образом придется опускать минимальную планку вашего дохода и работать с более простым для вашей головы проектом. Но есть еще вариант разобраться с тем, что обычному новичку не под силу, то есть хорошее знание предметных областей. В этом случае не придется бороться с толпой дилетантов, желающих отобрать ваш хлеб.

Очень интересным и отчасти забавным был пункт про собеседования. Эти пояснения фраз, которыми так и кишат объявления о работе, не могут не радовать. Вот пример:

«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.

«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.

Еще было сказано, что теперешние тесты на техническую часть не позволяют толком проверить реальные способности испытуемого, а лишь проверяют знания синтаксиса или какую-то странную логику искать правильный ответ в запутанных дебрях непонятно кем написанного кода, после прочтения которого хочется выключить компьютер и не включать никогда. Не самый лучший вариант, но что поделать.

И еще пару слов о поиске работы. Ваше резюме должно быть максимально полным с минимальным количеством слов. Всю основную информацию о вас вы должны изложить на первой странице, а технические навыки и умения расписать на остальных страницах. И никогда не врите. Это чревато сильными осложнениями в дальнейшем поиске работы, особенно если вы уже высоко забрались. И еще, важно быть приветливым с персоналом по найму, а то кто знает, по каким «официальным» причинам вы не подойдете в очередную компанию.

Тема была не сильно объемная, но несколько интересных моментов все же нашлось.

Первый момент — интересная мысль о том, что можно было бы делать программы, как аппаратуру, то есть собирать из компонентов. Разумеется, эта идея полностью не осуществима, ведь предугадать все возможные потребности попросту невозможна, однако программисты стараются писать многократно используемый код. Будем считать, что это и будут компоненты в будущих программах.

Еще одна тема — полное тестирование всех возможных исходов. Я раньше и не задумывался о том, насколько запутанным может быть путь от начала программы до ее завершения и сколько разных вариаций входных и выходных данных может принимать и отдавать программа.

В этот раз некоторые темы были, возможно, интереснее, чем в прошлой части. Но обо всем по порядку.

Часть «О карманных монстрах» — полезна тем, что направляет на изучение чего-то нового с каждой новой целью. Ведь, если все делать по накатанной, то: 1) мало прогресса, 2) решение довольно громоздкое и неизящное. Почти наверняка уже кучу программистов до вас столкнулось со схожей задачей и, если не было изящного решения, придумали новые возможности/утилиты/фреймворки, которые значительно упрощают процесс реализации идеи вашего решения. Еще раз показывается, что наша профессия подразумевает постоянное обучение. Запомните это.

По поводу идеи веб и использования браузеров для кроссплатформенноси приложений — рассказаны основные идеи, возникшие проблемы, одно из решений: отдать весь проект компании Google, с фразой: «уж они-то точно разберутся», что само за себя говорит о несостоятельности и некомпетентности сотрудников, да и звучит забавно.

В заключительном пункте этой части замечается, что для комфортного пользования вашим проектом/веб-ресурсом важны следующие факторы:

- Легковесность интерпритатора (либо среды времени выполнения)
- Кроссплатформенность
- Интерактивное отображение информации, для того, чтобы заинтересовать пользователей

А помимо этого идет рассуждение, что было сделано и как было сделано. Не думаю, что это стоит того, чтобы включать в данный конспект.

Несмотря на то, что ООП сейчас очень популярно и используется повсеместно, оно не такое простое, каким кажется на первый взгляд. Для того, чтобы получать максимальный результат, как это предлагают разработчики ООП языков программирования, необходимо построить идеальную модель решаемой задачи, чтобы можно было грамотно и корректно перенести ее в виртуальный мир.

По поводу ORM, мне нравится идея проецировать классы из ООП языков в базы данных без особых проблем. Это позволяет получить меньше проблем с эдаким конвертированием одного типа информации в другой. Но, безусловно, те, кто пользуется ORM технологиями, должны знать, как это все работает на более низком уровне.

Интересно, что автор много раз повторяет, что надо рефакторить код. Конечно, все понимают, что это очень важно, особенно на долгоиграющих проектах, однако многие попросту забывают/забивают. А затем сами на себя ругаются, когда пытаются добавить новый функционал, на основе уже имеющегося.

Честно говоря, эта часть книги мне не сильно понравилась. Тут больше проводилось какое-то сравнение неизвестных мне технологий, часть из которых устарела. Однако сами формулировки тем толкнули к следующим размышлениям.

Облачное вычисление. Очень популярная и используемая технология. Можно сказать, что все, что связано с Интернетом, относится к ней. Ведь архитектура такова, что вся логика выполняется, в большинстве своем, удаленно на сервере, а нам приходит результат. Это стало совсем обычным делом, ведь сегодня скоростной доступ в Интернет не является чем-то заоблачным.

По поводу прогресса технологий, надо подходить к этому делу не только с финансовой точки зрения, но и думать о совместимости своих версий. В книге приведено достаточно примеров крупных компаний, которым, к сожалению, не удалось реализовать решение этой проблемы. Также автор как бы высмеивает ту частоту, с которой некоторые компании выпускают новые версии своих продуктов. Понравилась фраза: «...можно

перескочить со второй версии сразу на четвертую, минуя третью и третью с половиной. Правда, уже анонсирована пятая».

В заключение приводится пример того, что все хвалят Windows за лучшую совместимость с периферией, чем у Linux. Однако потом выясняется, что у Linux дела обстоят лучше. А все из-за той же проблемы, о которой говорилось ранее. Как говорил очень хороший человек: «Думать надо», чтобы не было трудностей.

Лично мое мнение, будущее за Linux, ведь темпы развития и свобода распространения заметно выделяют этот вид ОС. Ну и одна из важных особенностей — все везде работает с разными версиями. И это хорошо :)

Проектирование информационных систем является очень трудным и важным делом. Разделение всей логики на уровни — очень грамотное решение. Во-первых, модульность — хороший подход для сопровождения проекта. Во-вторых, можно нанимать специалистов в конкретной области для достижения наилучшего результата.

Мне понравился блок о концептуальном устройстве. Я и не задумывался о том, сколько различных вариаций «звеньев АИС» может быть. Очень интересные подходы с толстым и тонким клиентами, а также подход об их комбинировании для того, чтобы получить плюсы каждого из них.

Самое важное — пробовать сделать что-то самому, а не смотреть примеры и только по ним работать. Важно понимать, как все устроенно изнутри, как работает каждая деталь, каждый механизм. А уже потом на основе имеющегося опыта подбирать оптимальные архитектурные решения.

Автор считает, что «шаблонное мышление» — это нечто плохое, что не дает самим людям думать и находить решения. Я думаю, что он не прав в том, что не совсем точно определил это понятие, касательно шаблонов проектирования.

Как говорится в негласном слогане паттернов: «Вашу задачу уже кто-то решал до вас», а это значит, что уже придумано элегантное решение, которое вам необходимо дополнить и свести к вашей проблеме. Тут, как говорится: «предупрежден — значит вооружен». Я считаю, это полезное знание.

Однако, стоит отметить разумность фразы, что лучше паттерны изучать не в самом начале своей карьеры программиста, чтобы на себе ощутить всю глубину идеи, чтобы понять, что ты, возможно, не учел.

Далее по книге идут рассуждения о пригодности шаблонов и «правильной», по мнению автора, идеи ООП. Тут он просто говорит, что шаблоны стали ненужными, что можно работать и без них, даже призывает отчасти к этому. Я не согласен. Шаблоны проектирования могут заметно упростить понимание и реализацию происходящего.

А вставка про сборку мусора была, как мне показалось, ни к чему. Да, глава посвящена тому, что надо думать головой, но это не причина для того, чтобы еще раз говорить, что сборщик мусора вещь хорошая, но «настоящие программисты сами следят за памятью». Каждый работает, как ему удобно.

Наиболее интересной мне показалась часть про UML диаграммы и средства для работы с ними. Мне кажется, что это весьма удачное решение, несмотря на критику автора, которое позволяет как бы отдаленно взглянуть на всю картину нашего приложения. Там ясно показывается какие части есть, с кем они связаны, что умеют и так далее.

Так же отметилась мысль, что в большинстве случаев при построении моделей мы наблюдаем за явлениями, хотя должны наблюдать за сущностями. Вот и получается, что мы видим все только с одного определенного ракурса. А это может привести к тому, что мы сделаем не универсальную модель или, что еще хуже, вообще неверную.

Эта часть книги также была полна всяческими историями из богатого опыта автора. Поэтому конспект писать было несколько затруднительно (особых размышлений истории не вызвали). Однако были интересные моменты, за которые хочется зацепиться.

Первый момент — это то, что темпы выпуска нового программного обеспечения все возрастают. Таким образом опытных и компетентных в одной сфере/среде разработчиков становится все меньше. А это серьезная проблема, т.к. именно из-за этого страдает качество. Поэтому я бы посоветовал, особенно новичкам, при выходе очередного нового инструмента или же среды не кидаться на нее, а лучше разобрать полностью все актуальные возможности предыдущей системы, а после браться за новое. Исключением может быть разве что случай, когда прежний инструмент уже пора поставить на полку и забыть о нем.

Еще был интересен раздел про сокрытие архитектуры. Очень поразил тот факт, что данные обрабатывались быстрее (а там их было ого-го сколько), чем читались и гуляли по слоям на пути к конечному пользователю. Все-таки в настоящее время действительно не хватает знающих специалистов, это подтверждает вставка автора из выступления на TechDays от компании Microsoft: «Последние годы я вижу тотальное падение компетенции в области баз данных. DBA, проснитесь!»

Смысл этой части заключается в том, что каждый уважающий себя и других программист должен поддерживать код проекта в чистоте. Речь даже не столько о клин-коде, хотя это очень важно, сколько о «чистке» старых версий реализаций и функций.

А смысл этого в том, что таким образом нам проще будет общаться с кодом. Мы не будем отвлекаться лишний раз на раздумья по поводу того, почему раньше было так, не будем искать наши закомментированные строки среди множества старых строк, а также никто не узнает о том, как плохо мы реализовали ту или иную функцию n лет назад :)

Еще слово было сказано по поводу тестов для систем, которые склонны часто меняться. На мой взгляд, тесты должны быть фундаментальны и проверять общую и основную логику приложения, а не какие-то мелочи. Потому как именно на перекраивание тестов под эти мелочи уходит больше времени, чем на написание работающей программы с учетом новых требований и пожеланий.

Опять же часть начинается с «историй из жизни», однако некоторые новые и интересные мысли проскочили.

Первая из них — автоматизация написания кода по прототипу (модели) проекта. Это весьма удобный инструмент, особенно если объемы столь велики, что эта рутина заставляет тратить слишком много полезного времени. Но опять же и он не идеален, а именно возникают проблемы, когда мы начинаем смешивать модель и ручное написание кода поверх сгенерированного. Хотя я слышал, что современные системы более проработанные и функционал существенно улучшился. Лишний раз подтверждается мысль, что лень — двигатель прогресса.

Вторая зацепка — генерация кода по моделям. Идея схожая, однако мы не набрасываем элементы на панель, а пишем конфигурацию модели в XML. Как утверждает автор, мы можем 600-ми строчками XML кода заменить около 30.000 строк кода на SQL, Java, XML для разных слоев нашей архитектуры. А также написанный таким образом код, вернее сгенерированный, не нуждается в тестировании. Как итог, наша производительность возрастает многократно.

Опять же приведены истории из жизни. Очень отчетливо видно, насколько важна качественно продуманная архитектура. Ведь именно от того, как она разработана и реализована зависит удобство и простота дальнейшего расширения продукта. Мы не получим (или получим с меньшей вероятностью) ошибку в модуле, когда изменяем совершенно другой. А также необходимо своевременно тестировать приложение. Благодаря этому можно быть уверенным, что все работает корректно, также в долгосрочном периоде. Особенно остро стоит эта проблема в низкоуровневых языках, где разработчик сам «трогает» память.

В целом книга получилась наполовину техническая, наполовину художественная. Возможно, это правильный выбор, чтобы не утруждать читателя большим количеством технических подробностей, чтобы он мог немного отдохнуть и «почитать для души». Истории из жизни были наполнены смыслом, чтобы мы смогли в дальнейшей работе отдаленно вспомнить неудачи из рассказов и заранее их устранить.

Возможно, она будет интересна «бородатым дядькам», у которых тоже есть свои истории. Им будет интересно узнать, что было раньше, как и что использовали на реальных проектах. Сравнят, изменилось ли что-либо в настоящее время.

Борисевич Павел

О нашей профессии

Из всей информации в главе “О нашей профессии” наиболее полезными мне показались советы автора начинающим программистам в поиске вакансий. В принципе, можно согласиться с мнением автора, но полностью полагаться на его советы не стоит. Как мне кажется, не всегда следует соблюдать краткость в своём резюме, если ты начинающий программист. Нужно максимально понятно объяснить с чем ты уже сталкивался и какие задачи ты можешь решать, чтобы потом не возникло недопонимание. Когда у программиста появится достаточно большой опыт работы, тогда, наверно, ему уже не обязательно стараться подробно рассказывать о всех своих приобретённых навыках, с опытом он будет понимать, что необходимо указать в резюме, а какая информация будет лишней.

Можно ли конструировать программы как аппаратуру?

Из всего прочитанного материала наиболее интересной мне показалась тема использования конечно-автоматной модели для программного компонента в софтостроении. Эта модель, как мне кажется, не самая удобная, и пользоваться ей можно только для решения небольшого класса задач, при этом время отклика компонента оценивается приближённо, а тестирование в большинстве случаев происходит выборочно.

ASP.NET и браузеры

Самой интересной информацией из прочитанного материала мне показалась идея универсального программируемого терминала, которым является веб-браузер, поддерживающий стандарты взаимодействия с веб-сервером. Оказалось сложным разработать приложение, корректно работающего хотя бы под двумя типами браузеров.
А тестировать приложение нужно не только под разными браузерами, но и под разными операционными системами. С учётом версий браузеров. Поэтому к началу 2000-х годов в корпоративной среде появился фактический стандарт корпоративного веб-приложения: Internet Explorer 6, который не плохо выполнял свои функции, до того, как приходило время переходить на новую версию браузера. Тогда выяснилось, что их новые версии далеко не всегда совместимы с находящимися в эксплуатации системами. И по этой причине простое обновление Internet Explorer 6 на 7 могло вызывать паралич информационных систем предприятия.

ООП – неизменно стабильный результат

До того, как я прочитал этот материал, я не видел никаких недостатков в ООП. Мне казалось, что ООП – это самое удобный и вообще правильный способ программирования. Раньше я не задумывался о тех проблемах, которые могут возникнуть, если использовать ООП в больших проектах. Действительно ведь возникает много трудностей: невозможность быстро разобраться в чужом коде, наличие многих неявных зависимостей, человеческий фактор становится ключевым риском проекта. На практике объектно-ориентированный подход в большинстве случаев превращает проект или продукт, переваливший за сотню-другую тысяч строк, в “Ад Паттернов”, который, никто без помощи авторов развивать не может.

Прогресс неотвратим

Из прочитанного материала я решил выделить наблюдения автора за некоторыми продуктами Microsoft. В целом все они критикуются. Во многом, наверное, можно согласиться с этой критикой. Например, стремительные обновления NET ставит в трудное положение разработчиков, которым приходиться преодолевать появляющиеся риски и тратить средства на переобучение персонала. Обновленная версия SQL 2005-го года оказалась не доработанной, приложения стали работать медленнее. Windows Vista не оправдала ожиданий и оказалась невостребованной с появлением “семёрки”, так и не успев занять сколь-нибудь значительную долю парка “персоналок” и ноутбуков. Microsoft Office критикуется за смену своего интерфейса, который, как мне кажется, не такой уж плохой, чтобы требовать возможность его изменять.

Слоистость и уровни

Наиболее полезной информацией из прочитанного материала мне показалась классификация слоёв и уровней корпоративной информационной системы, что важно знать не только разработчикам, но и тем программистам, которые будут наращивать на неё свои приложения. Есть несколько точек зрения, с которых можно рассмотреть любую автоматизированную информационную систему. В них можно выделить слои, кроме этого даже в самых простых программах различают как минимум два уровня. Если мы наложим слои системы на её уровни, то получим достаточно простую матричную структуру, позволяющую бегло оценить, какой из элементов необходимо реализовать своими силами или же адаптировать уже имеющийся готовый.

Ultima-S – КИС из коробки

Наиболее полезным мне показался рассказ про КИС Ultima-S, над которой трудился автор. Интересно было узнать о двух основных подходах в разработке КИС, а именно «от производства» и «от бухгалтерии». Ultima-S являлась примером подхода «от бухгалтерии» как и многие другие разработки КИС того времени. Эта система функционально была хорошо обеспечена для своего времени, но не получила широкого применения всех своих технических возможностей, которое наступило спустя десять лет в других системах. Главной проблемой проекта Ultima-S и многих других, по мнению автора, было отсутствие маркетолога, определяющего вид продукта и программу продвижения, что сделало бессмысленным большое количество технологических действий.

Журнал хозяйственных операций

Из прочитанного материала я решил выделить раздел про учётные приложения, в котором рассказывается про такой антишаблон как «таблица остатков». Суть его заключается в том, что приложение использует лишь остатки, не принимая во внимание историю их изменения, что приводит к некоторым сложностям. Надёжнее использовать в приложении журнал операций и вместе с этим хранить вычисленные остатки. Хранить таблицу остатков следует для того, чтобы не ждать окончания их расчётов, которые могут затянуться.

Архитектура сокрытия проблем

Из прочитанного материала я решил выделить рассказ автора, где главным героем была крупная фирма-разработчик, которая занималась созданием ERP-систем для розничной торговли. Через какое-то время у компании появились проблемы разрастания баз данных: время отклика как интерактивной работы, так и пакетов увеличилось, производительность понемногу деградировала. Найти грамотное решение не получилось, смена СУБД на более мощную и более производительное «железо» не привело к результату. Спустя долгое время попыток найти выход из сложившейся ситуации, компания была поглощена холдингом Cegid за свою некомпетентность в области СУБД.

Тесты и практика продуктового софтостроения

Из прочитанного материала я решил выделить рассказ автора, про модульное тестирование. Начинается всё с причин, по которым использовать модульные тесты для тиражируемых веб-приложений нецелесообразно. Наверное, главные причины: сильное затруднение рефакторинга и отсутствие необходимости выпускать обновления слишком часто. Была затронута интересная проблема: в некоторых случаях соотношение тестирующего кода значительно превышало объёмы рабочего. В отличие от функциональных тестов, завязанных на интерфейсы подсистем, модульные тесты требуют переработки одновременно с рефакторингом рабочего кода. Это увеличивает время на внесение изменений и ограничивает их масштаб.

Программная фабрика

Из прочитанного материала я решил выделить тему: программная фабрика, в которой рассказывается про инструменты CASE, позволяющие генерировать код, скомпилировав который, можно сразу получить работающее приложение. Первые CASE инструменты, выросшие из редакторов графических примитивов, были представлены в 1980-х годах. Можно заметить некоторые недостатки CASE-средств:

1) Если ручное написание кода принять за максимальную гибкость, то CASE может навязывать стиль кодирования и шаблоны генерации частей программ.
2)CASE работает только при условии, что ручные изменения генерируемого кода исключены или автоматизированы.
В качестве решения перечисленных проблем появились так называемые двусторонние CASE-инструменты, позволяющие редактировать как модель, непосредственно видя изменения в коде, так и, наоборот, менять код с полу или полностью автоматической синхронизацией модели.

Что-то с памятью моей стало

Из прочитанного материала я решил выделить тему с забавным названием “Что-то с памятью моей стало”, в которой рассказывается про утечку оперативной памяти сервера. Программа, создавшая эту проблему, забирала около одного гигабайта памяти в сутки, причём никак не возвращая ее обратно даже после отключения всех пользователей. Таких неприятностей можно было избежать, если бы в процессе разработки ПО проводилось тестирование. Но в результате совместных усилий проблему решили, хоть и остались некоторые другие.

Гетьман Святослав

Технологии

Женщины в IT

Снижение количества женщин в отрасли из-за «резко» возросшей популярности универсальных сред – лишь одна сторона медали, затронутая автором. На мой взгляд, женский пол с лёгкостью способен разбираться в сложных математических абстракциях и работать в более универсальном пространстве, недели в коридоре узкоспециализированных программных сред. И ссылаться на половую принадлежность, как это делает автор, грубо.

Почему же тогда девушек меньше? Увы, к сожалению, далеко не каждую девушку в школе и дома обучают так, чтобы к университету у неё было не просто образно-гуманитарное представление мира, но и формализовано-техническое, которое позже позволит ей стать специалистом в IT-сфере и без труда справляться с теми проблемами, с которым справляются мужчины.

Системы управления версиями

На дворе 2015-й год, и мной одолевают очень сильные сомнения, что за пределами университетов есть курсы и / или предприятия, которые пишут рабочий и окупаемый продукт, не используя Git или другие ревизионные системы.

О заказчиках

Хочется по своей наивности верить, что далеко не все заказчики представляют из себя «большой глупый кошелёк», как это с одной стороны было продемонстрировано в главе «Диалог о производительности». Уверен, что есть компании, которые «разрабатывают сами на себя» (к примеру, EPAM и гиганты вроде Google, Microsoft, Apple), где заказчиком выступает человек из компании (CEO).

Битва фреймворков

Битва фреймворков и узкоспециализированных сред разработки мне не совсем ясна. Точнее, не ясно, почему этот вопрос стоит так остро. Возможно, это оттого, что я сам использовал фреймворки для написания некоторых программ. Но на мой взгляд, надо чётко понимать, что где-нибудь в NASA или на военных разработках, они используют узкоспециализированный софт, чтобы проводить более точные расчеты. Не исключен также и вариант написания той же NASA своего программного решения (если хотите, то это будет фреймворк), которое позволит им сделать производительность повыше и точность побольше (зависит от исходной цели разработки данного продукта). Затем этим фреймворком могут воспользоваться другие компании, которым необходим аналогичный функционал и которым необходимо экономить время с деньгами на разработку. Плохого в том, чтобы не тратить время на «создание велосипеда», я не вижу (только если это не университет, где написание фреймворка даёт пишущему чёткое анатомическое представление того инструмента и последующих, которые он будет использовать).

Веб-разработка

Не понравилось, что веб-разработка автором едко задвигается в далёкий тёмный угол, дабы не мешала разрабатывать и развёртывать настоящие полноценные софтины для декстопников и тому подобных спецов. Складывается не очень хорошее впечатление, что автор просто не ужился с новой ~~женой~~ технологией. Если не можешь мириться с тем, что времени стало уходить больше на какие-то мизерные проблемы, то пора уходить из отрасли. Хотя и опасения автора тоже понятны, но, скорее, с точки менеджера: времени действительно стало уходить больше на всякую «мишуру» и «конфетти», особенно это видно в чётком разграничении должностей и сфер, типа «разраб» и «дезигнер».

Как ни крути, но, увы, нельзя вот так просто взять и забыть о сфере веб-разработки даже на микросекундочку, ибо рынок уже как с 2008 года дядюшкой Стивом Джобсом построен и заточен под мобильные технологии, которые направлены на максимальное удобство и (что вполне логично, исходя из названия) мобильность. Есть тут и подводные камни в плане безопасности пользователя, но,

как оказалось, их предостаточно и у разработчиков, которым эта технология была в новинку. Об этих камнях автор достаточно подробно рассказал.

В утешение будет сказано, что софт для десктопов и суперкомпьютеров до сих пор спонсируется и разрабатывается, изменилось лишь отношение к нему, как, впрочем, и к стационарным ПК с ноутбуками. Всем захотелось лёгкости и мобильности. А также воздушности и облачности. Посему для ПК/ЭВМ/техники разрабатываются либо какие-нибудь профориентированные штуковины (банк. софт для АТМ, игрульки на XBOX и т.д.), либо урезанные игрушечные шарики с рождественской ярмарки веб-серверов и веб-клиентов.

Благо сейчас время действительно другое (бежит уж очень быстро; книга написана в 2013м, а на дворе заканчивается 2015й, по прикидкам, информация успела возрасти в раз 16 – в два раза за каждое полугодие). Да и получение знаний благодаря курсам уже не кажется таким кощунственным, как и сама веб-технология со всякими дебрями вроде скриптовых языков, разных браузеров и развёртываемости на всех устройствах от планшета до микроволновки. Можно точно сказать, что ситуация стала обратной: с вебом многие уже на «ты», а вот как софтину написать на плюсах знают только бородатые «битарды» с пивным от опыта животом.

Изменится ли что-нибудь в ближайшее время?

Да, если технология начнёт массово давать сбои и ставить нас в неловкое положение из-за скрытых и / или забытых уязвимостей самого веб-программирования. И нет, если веб-разработку доведут-таки до ума.

Об авторском подходе

Всё меньше и меньше начинает нравится подход автора к новым веяниям, начинает чувствоваться субъективная неспособность перестроиться. Конечно, это, с одной стороны, даёт массу тем для дискуссий, с другой стороны, формирует однополярное мнение, какую-то избитую категоричность, из-за которой проблема хоть и разобрана, но только с одного боку.

О взгляде автора на молодое поколение программистов

Прежде всего, меня возмутил тот факт, что в глазах автора практически вся молодёжь не знает SQL. Конечно, у него опыт больше, чем у меня, да и людей и фирм он повидал больше, но, на мой взгляд, сейчас обстановка более-менее налажена, тот же конвейер и даже курсы уже следят за тем, чтобы выпускаемый полуфабрикат был закалённым, чтобы знал азы вроде Java / C++ / SQL / JScript / HTML.

Понимаю и соглашусь в то же время, что рынок программистов находится в странном состоянии, а именно: людей не хватает, но полуфабриката тьма. В принципе, если только таким взглядом и смотреть с ~~авторской колокольни~~ позиции автора, то ясно его негодование. Да, супер-пупер специалистов-профессионалов готовят исключительно в вузах, и то там есть свой отбор-сито, через которое проходят далеко не все.

Но, на мой взгляд, это уже проблема человека, что он чего-то не знает или не прошёл отбор. Или на курсы не попал. Плюс никто не отменял такого правила: кто хочет, всегда сможет. Так работает Вселенная, по такому закону работаем и мы.

Об ООП

Касательно позиций насчёт ООП тоже согласен не полностью, ибо паттерны для того и созданы, чтобы облегчить построение более сложных систем. Да, правильно было замечено, что во многом, если не во всём, они представляют кальку с природы, перенесённую на связи и принципы ООП. Опять же, мы довольно ограничены в технологии (а может и фантазии), чтобы воплотить в жизнь мечту автора (и не только его): сделать связи красивее, элегантнее и настолько проще, чтобы можно было не запутываться в построении сложных систем. Пока что никуда мы не денемся от понимания на уровне паттернов, однако, я считаю, что не за горами языки или системы, в которых реализация паттерны будет проводиться так, что нам не нужно даже знать особую структуру в зависимости от случая, просто написать пару инструкций.

Насчёт СУБД и ООП во многом понимаю автора, но должен сказать: таковы правила, надо по ним играть. Это здорово, когда программист знает SQL и запросто может написать крутые запросы, но, на мой взгляд, коль мы крутимся в сфере веб-разработки, где всё сплошь и рядом одни объекты, то, будьте так добры и любезны, также уважительно относиться к ООП и как следствие оборачивать взаимодействие объектов / моделей с базами данных в ORM.

Это не значит, что не стоит придумывать новые схемы. Стоит. Но, как очень важно подмечено Тарасовым, необходимо наличие математической теории, которая это дело позволяет обобщить. А это важно, в особенности, когда размеры растут и необходимо искать какие-то закономерности.

Методология Scrum, уродование отрасли

Ещё позабавило во многом отношение автора к методологии разработке SCRUM. Вот уж чего я не ожидал, так этого такого едкого сарказма! Да и в адрес весьма популярной штукенции, которую никто мимо не пропустит! Да, во многом это верно, ведь это затормаживает процесс, добавляет больше пустословия и бюрократии, которая продукт к завершению не движет. Да и денег, я уверен, на этот самый «стыд-и-скрам» тратится больше, чем если бы ребята им не пользовались.

Конечно, это дело вкуса и во многом престижа. И, на мой взгляд, то, о чём кричит Сергей Тарасов в этой главе, так это о том, как напускные аспекты вроде того же престижа, влившиеся в понятие «разработки программного обеспечения», изуродовали отрасль, в которой начали прорастать очень ценные зёрна (ООП, СУБД, паттерны и т.д.).

Взгляд, конечно, весьма консервативный, но он определённо заслуживает внимания.

ВКЦП в облаках

Вычислительная мощность ПК не используется на 100%.

Неудивительно, ибо рядовой пользователь выполняет с помощью ПК достаточно нехитрые задачи. В тех случаях, когда он выполняет что-то сложное, что нагружает ЦУ и ОЗУ, то, во-первых, стоит задуматься, а является ли рядовым данный пользователь (может быть, это программист вроде Константина Антоновича Зубовича, которому захотелось «поиздеваться» над программой, используя её слабые места), и, во-вторых, делает это всё тот же рядовой пользователь неявно, с помощью вспомогательных средств. Это, в частности, относится к геймерам: они ведь не производят ручные расчёты на матрицах, не влезают в дебри линейной алгебры и теории операторов, чтобы получить ту картинку и тот функционал, который у них есть благодаря тому, что кто-то из числа оговоренных выше лиц (программистов и не только) уже со всей этим достаточно «накувыркался». Хотя, надо признать, игра, провоцирующая людей на дотошное копание ради мало-мальского результата для меня есть очень даже неплохая идея для развивающей игры (для студентов ПМ) ☺

Из-за избытка мощностей на узлах (в случае распределённой системы) происходит переход от распределённых систем в пользу централизованных.

И я считаю, что это правильно, что, если есть излишки в системе, которыми никто не пользуется (опять-таки, надо понимать, когда это «никто» не такое уж и пустое множество). Далее автором книги будут приведены доводы в пользу централизованных систем, в частности, речь пойдёт о ВЦКП и её сути. Во многом, с этими доводами, которые как все дороги ведут к Риму в виде «ВКЦП первым пытался реализовать взаимовыгодную систему облачных вычислений», я согласен и считаю, что шаг этот, принятый и НИИ, и, частично, государством, был верный.

ВКЦП = Вычислительный Центр Коллективного Пользования

На мой взгляд, тесная связь с ныне популярными «облаками» очевидна. И для советской страны в 1972 году первым сделать такие прорывные попытки, при этом затронув важную связь между экономикой, государственной сетью и объединением региональных центров, было просто великолепным подспорьем, которое, глядишь, могло бы сделать много полезного шума и предоставить конечному пользователю (да и государству тоже, об этом не забывать) много удобств.

Суть ВЦКП со временем не изменилась, сменилось лишь название.

Согласен. Опять же, надо видеть в ВЦКП скрытую систему работы с центральным сервером-«облаком» и все вопросы и сомнения отпадут: да, это то, чем сейчас интересно заниматься очень большому количеству программистов. В том числе и мне интересны такие подходы.

ВЦКП пролетело:

1) **Профуканная миниатюризация**
2) **Просчёты и развал страны**

В точку. Я бы ещё добавил в этот список самое по себе понятие «лихие 90-ые», когда работать надо было за еду, когда во все сферы жизни стал вмешиваться рынок, отчего налаженная до того система «мороженное за 3 копейки» превратилась в чёрт знает что. Понятно, что рано или поздно в программирование примешалась бы экономика, расширив и размыв понятие, но то, что это у нас произошло так бурно и стремительно (без подготовки, как на Западе) ещё раз, на мой взгляд, доказало, что ~~новые русские не сахар~~ стоит уделять время прогнозированию будущих решений и всегда держать ухо востро.

Возврат к централизованным системам?

Возможен. И я даже знаю почему.

Это предлагает тем же геймерам, а значит, одной из самых «жирных» прослоек современного инфообщества неописуемые возможности, равно как и прибыль таким корпорациям, как Sony, Nintendo, Microsoft. Если подробнее, то не секрет, что та же Sony ведёт разработки «игровых подписок» для конечного пользователя, что из себя представляет тот же централизованный подход, но уже по отношению к играм. Фактически, игрок будет играть в одну и множества доступных за определённую плату терминалом «игровых проекций», которые запущены / поддерживаются сервером-«облаком».

Плюс, эта система уже активно используется, вспомнить хотя бы банковское дело. Программисты в банковской сфере не только много получают за прогнозирование рынка, но и за создание таких централизованных систем и поддержку безопасности для терминалов, коими и являются небезызвестные банкоматы.

На мой взгляд, за этими подходами будущее. Но всегда стоит помнить, что даже у такой схемы есть скрытые слабые места как банальное падение серверов и, как следствие, пропадание всего функционала на терминалах. Как это обойти – время покажет.

Затратность программирования распределяемого приложения, но его выигрышность в сравнении с централизованными приложениями.

На мой неподготовленный взгляд, затраты и там, и там. Где больше судить не берусь, ибо толком не работал ни с одним из направлений. В плане выигрышности, на мой взгляд, стоит опять же вернуться к игровой сфере (Остапа Бендера понесло ☺) и взять, к примеру, мультиплеерные игры.

Тяжелы ли они для разработки? Ну да, надо попотеть над хорошим мультплеером больше, чем над обычной синглплеер игрой.

Интереснее ли пользователю мультиплеер или синглплеер? Здесь чёткого ответа нет, есть только статистика, которая показывает, что с поднимающимся уровнем технологий, используемых в игростроении, мультиплеер всовывается практически во все игры, а это значит, что да, пользователю это «необходимо». Если смотреть более объективно, то это дело извращённости и вкусов конечного пользователя, ему, во всяком случае, необходимо предоставить право выбора.

Так, основываясь на игровой индустрии, можно сделать выводы и обобщения для других сфер.

В заключение скажу, что вот в чём я точно уверен, так это в том, что разработка приложения, которое выборочно совмещает в себе и распределённость, и централизованность, очень сложный процесс,

который нуждается скорее в какой-то теоретической подоплёке для дальнейшего упрощения, нежели в супер-росте технологий.

Работа с локальной копией данных центральной БД.

Решение отличное, особенно для банков, но, хотелось бы верить, далеко не единственное.

SaaS и data manipulations

SaaS = software as a service

Хороший вопрос: «софтверные фирмы производят услуги»?

Хороший вопрос на подискутировать.

Соглашусь с точкой зрения автора: услуги не производят, а ОКАЗЫВАЮТ. А фирма производит ПРОДУКТ, а не услугу. Разница между ними в том, что продукт можно хранить, он физически осязаем, и время и место его потребления могут растягиваться, чего не скажешь об услугах: оказал, и она тут же «нашлась к месту».

Если уже принять тот факт, что фирмы = продукт, то без схем, описанных автором, не обойтись.

1. Производитель создал продукт, продукт обязательно проходит по рукам дистрибьюторов (консультанты, продавцы и т.д.) до конечного пользователя.
2. Программа как услуга сразу доходит до конечного пользователя, минуя дистрибьюторов (единственный это сам Интернет). Если пользователь программного продукта работает с ним через ВЦКП, то последний предоставляет доступ пользователям к налаженной системе.

Из чего следует, что в первом случае «навар» и проценты приведут к излишней дороговизне продукта, который может быть не до конца отлаженным, а во втором случае есть риск краха всей системы ВЦКП. Но и надо отметить, что рынок добрался и до второго пункта: появилась цифровая дистрибуция, люди за это получают денег не меньше, чем за обычную, «аналоговую».

От CORBA к SOA

CORBA – Common Object Request Broker Architecture

SOA – Service Oriented Architecture

Бурные 90-ые, шум вокруг понятий

что автоматически приводит нас к уже ранее затронутой теме «почему облажались с ВЦКП». Так, отчасти, и тут, с CORBA и SOA. Конечно, рассудили здесь не «по понятиям», но по простоте и популярности.

Интероперабельность CORBA

Огого, поддержка одной архитектурой множества различных платформ и архитектур поменьше/повыше всегда меня восхищало и радовало. Не знаю, стоила ли эта игра свеч, если она в итоге выдохлась. Надеюсь, что стоила, ибо судить о том, чего уже практически нет и с чем работать не придётся очень трудно и поверхностно.

Почему CORBA улетела с рынка?

1) **Долгий процесс согласования стандартов, отчего рождаются решения «на местах» (у Sun, к примеру)**
2) **Сложность кода (для Javистов)**
3) **Сложность спецификации + появившиеся упрощённые Java решения**
4) **Бум дот-комов и, как слдествие, дороговизна коммерческого решения**
5) **Раскрутка упрощённой реализации веб-сервисов в SOA**

По поводу пролёта с CORBA автор всё изложил довольно чётко, поэтому мне нечего добавить.

Sun могла объединиться с CORBA

И никто бы из двоих не улетел, все были бы в выигрыше и концепцию довели до ума. Да, согласен с этим.

SOA появилась как хорошее, но сырое подспорье CORBA

И её сырость заключалась, по словам, автора, в вынужденных надстройках для достижения базового для CORBA уровня и простоты.

Зачем тогда необходимо было пересаживать всех и вся на сырую, но новую землю – я не могу понять. Возможно, было просто дешевле, а это, как на зло, не такое уж и программирование, чистый маркетинг, поднасоливший индустрии. Или, быть может, не всё так прозрачно?

Из-за упрощения в SOA «порезали» функционал CORBA, а именно:

1) **Отсутствуют сессии и состояния в вебе**
2) **Переход от интерфейсов на C к XML.**
3) **Отказ от автоматической подгрузки связанных объектов (lazy initializing)**

1. -> отсутствует возможность транзакций -> из-за чего надо использовать шаблоны (ВСЕГДА И ВЕЗДЕ) -> more code, MORE CODE!

   У серверов нет состояния -> нет и объектов

2. *(Ох, XML ужасен, ужасен, ужасен)*
3. Тоже глупо, ведь плюсов у ленивой инициализации хоть отбавляй.

Плюсы CORBA: хорош, как многопоточный сервер, хорош как веб-служба.

Минусы CORBA: см. пункты выше, точнее, почему CORBA вылетела

Плюсы SOA: затмеваются минусами

Минусы SOA: сырьё, недовведенное до ума

Хотелось бы верить, что нас ждут другие решения.

Прогресс неотвратим

Вымирание больших ЭВМ

А где же ЦОДы и «облака»?

Большие ЭВМ не вымирают и не вымрут никогда, ибо они являют собой образцы стратегичности, требовательности в купе с защищённостью и производительностью. А за это многие дядьки с тугими кошельками отдадут денюжки.

Очень понравилась мысль: **«Солдаты от софтостроения не должны рассуждать, ибо генералы договорились с поставщиком».**

Где тогда грань между рабским трудом и профессией программиста?!

.NET

Из-за постоянных изменений, так активно производимых Microsoft (они там определиться не могут, или решают зарабатывать на «сыром и вялом»), .NET недозрелая платформа: концепции меняют как

перчатки, а это несёт убытков больше, чем дохода (курсы и т.д.). Однако, надо сказать, что тому же Microsoft это, наверное, выгодно, предоставлять возможность переобучения для новой платформы (т.е. всё-таки доход, но для Microsoft, а для компании, которй нужно сотрудников переобучить, это убыток).

Office '07

Нет вшитой поддержки старых версий и нету возможности выбора интерфейса (если только не проплатить). Да, я помню, лично на себе испытал эти проблемы и скажу, что Word 03 самый стабильный и лучший из MS OFFICE пакетов. LibreOffice не в счёт, всё-таки за него не надо платить и этот «леденец» пришёл ко мне с опытом, отчего я его ставлю, как и автор, выше остальных.

SQL Server

Деградация: унификация, торможение из-за .NET, упущение мелочей из старых версий, несовместимость форматов.

Vista

Потренировались на вас за ваши деньги – в точку!

Продажа долей Гейтса и Балмера – хм, хм, хм. Что бы это могла значить.

О материальном

X64 плох, драйверов на него нет и он отрезает путь к использованию старый версий и старого софта.

Есть немного, но поскольку уже 2015-й год, с этой проблемой уже разобрались и драйверов хватает. Другое дело, что ставятся они зачастую некорректно и нужно ручками это исправлять. Приятно ли это? Кому как. Мне, допустим, да, приятно, я люблю копаться, я от этого больше понимаю. А вот обычный пользователь по логике вещей пошлёт систему на три буквы.

И посылает. Но продолжает пользоваться ей в силу натренированной беспомощности взять и пересесть на MAC OSX / Linux.

Проектирование и процессы

Совершенство достигается не тогда, когда нечего добавить, а тогда, когда нечего убрать. (Антуан де Сент-Экзюпери)

Великолепный эпиграф. Эти слова без сомнения должны быть девизом уважающего себя проектировщика.

КИС = Корпоративные Информационные Системы

КИС претерпели развитие от закрытых монолитов до открытых / модульных систем, вместе с тем пожертвовав стандартизацией процессов и базовой функциональностью ради конкурентоспособности.

Ах, рынок, что же ты творишь?!

Краткий словарь начинающего проектировщика

Самый полезный (и весёлый) параграф из всех в этой главе

Задача проектировщика – поиск простоты.
Очень просто сделать сложно. Сложнее делать проще.

Сам словарик (не нуждается в комментариях):

- «Это был плохой дизайн». Это спроектировано не мной.
- «By design» (так спроектировано). Ошибка проектирования, стоимость исправления которой уже сравнима с переделкой части системы.

• «Это не ошибка, а особенность (not a bug but a feature)». Прямое следствие из «by design».

• «Это может ухудшить производительность». Не знаю и знать не хочу ваши альтернативные решения.

• «Нормализация не догма». Потом разберёмся с этими базами данных, когда время будет.

• «Это наследуемый модуль». Этот кусок со многими неявными зависимостями проектировали достаточно давно, скорее всего стажёры.

• «Постановка задачи тоже сложна». Ума не приложу, откуда возникли эти десятки тысяч строк спагетти-кода.

• «Сроки очень сжатые». Мы давно забили болт напроектирование.

• «Наши модульные тесты покрывают почти 100 % кода». А функциональными тестами пусть занимаетсязаказчик.

• «В нашей системе много компонентов». Установку и развёртывание системы могут сделать только сами разработчики.

Слоистость и уровни

Разбираться в КИС надо не только разработчикам самой КИС, но и программистам, которые собираются что-то «наращивать» на этой системе.

АИС = Автоматизированная Информационная Система. Состоит из:

Людей

Информации

Компьютеров

Устройство АИС с точки зрения проектирования делится на:

Концептуальное устройство

Логическое устройство

Физическое устройство

Концептуальное устройство

Используется анализ для составления устройства.

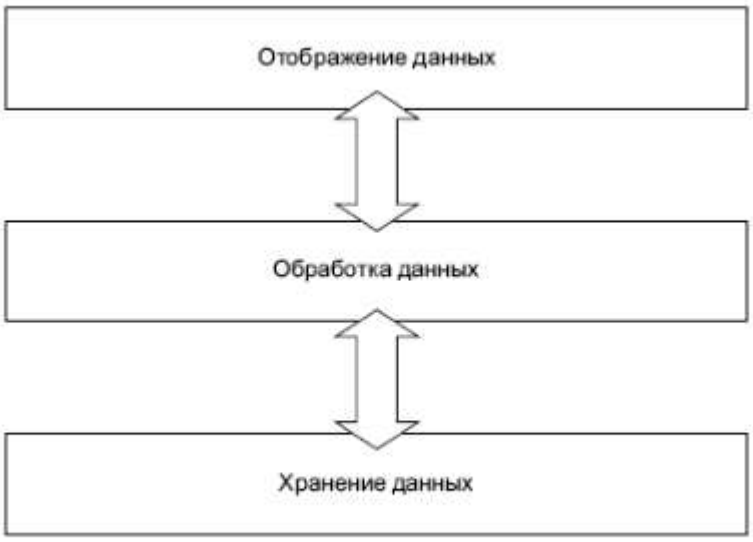


Рис. 4. Концептуальные слои АИС

Представленные слои существуют в ЛЮБОЙ ИС. Просто нужно уметь их искать.

Логическое устройство

Используется синтез для составления устройства.

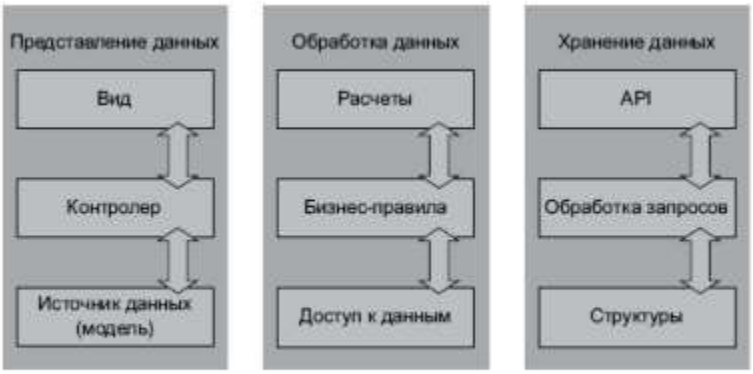
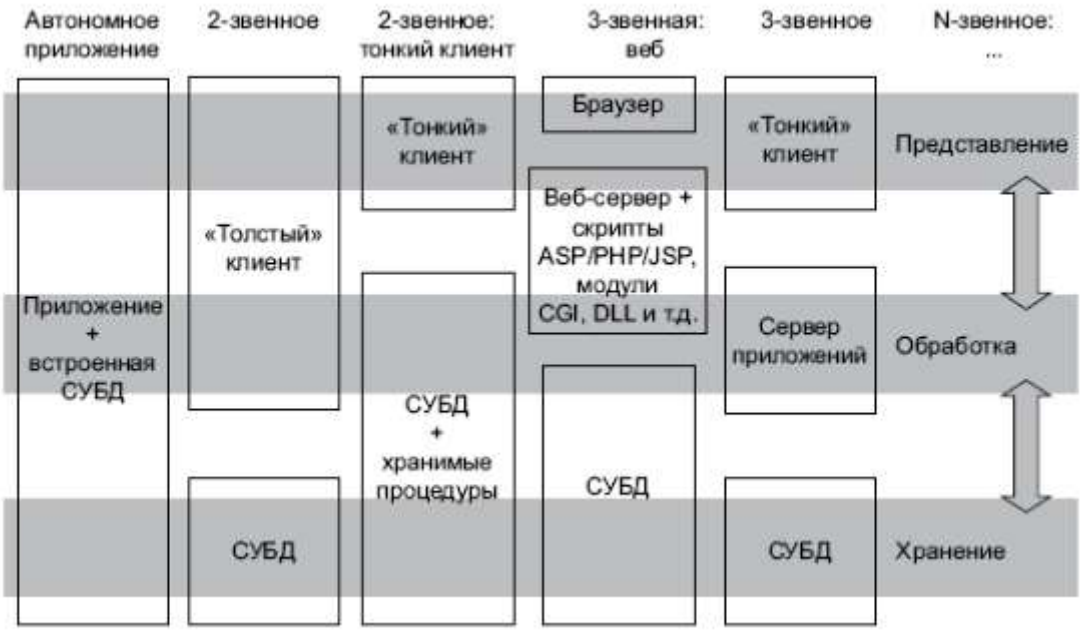


Рис. 5. Пример организации логических слоёв АИС

Нет строгого определения. Поэтому определение организации происходит путём ответа на вопрос «зачем нужен этот слой» либо использованием шаблонных решений и их подгонки.

Физическое устройство

Используется синтез для составления устройства.



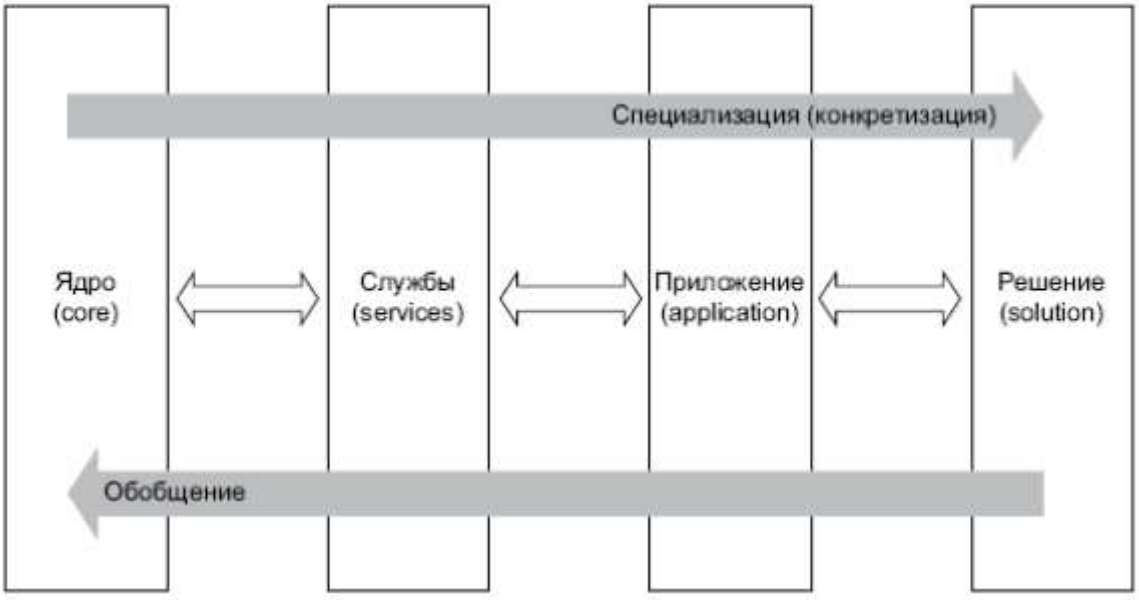
Thin Client – имеет только логику отображения (терминал / браузер)

Rich Client – прикладная обработка логики без сервера, сервер как хранилище

Smart Client – золотая середина между Thin и Rich клиентами

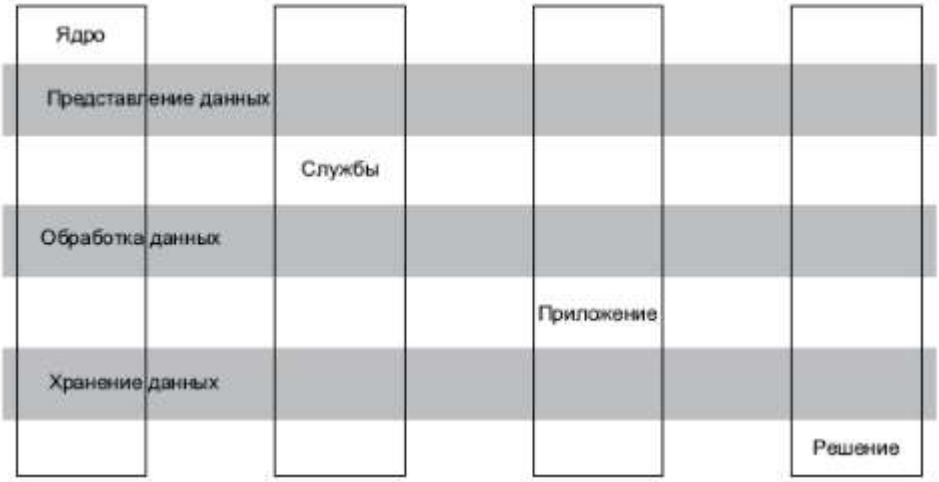
Насыщенное веб-приложение – Thin оснастка + возможности Rich

Уровни



Общие функции вынося в модули / API системного уровня. Функционал нескольких предметных областей выносится на уровень решения.

Совмещение



Многозвенная арихтектура

В концептуальном устройстве всего три части.

В физическом их может быть сколько угодно.

Доля специалистов по системному программированию падает -> архитектуру выбирают особо не думая, прочитав статью и не разобравишсь, либо «потому что иначе не умею».

В этом плане я вспоминаю Константина Антоновича Зубовича и его великие слова: «Что вы, Гетьман, по помойкам разным лазите? Лучше бы книжки читали». Само собой разумеется, что под помойками понимается СМИ и неподтверждённая информация, такая как заметка на блоге / форуме или пущенный слушок.

Как с этим бороться? Воспитывать интерес самообучения в людях, основанный на недоверии к глупостям, незнанию, поспешным и необдуманным решениям.

Прослойка парсинга информации достигает 80% от всей работы приложения, а ведь это пустой процесс.

Да, это пустой процесс, пока вы пользуетесь Google, который активно обрабатывает такие big data, что вам и не снились.

Практики управления «кейсами» в ИТ…

… и русифицированный клон западного приложения? Или же ещё один успешный проект?

Смотря, кто и как использует эти «кейсы». Если менеджер толковый и не зря ходил на лекции в студенческие годы, то он воспримет всё это как совет в рамках одной глобальной игры под названием «как утереть товарищу нос», ведь кейсы ничем не отличаются от популярных технологий: всплывают тогда, когда больше не нужны. Ну а если менеджер глуповат, то будет всё делать по «инструкции» и сам проект становится каким-то безжизненным.

Критичность проектов в ИТ падает, отчего решения переходят из технической в управленческую сферу.

Ищите возможности сократить путь информации от источника к пользователю и обратно.

Архитектура не есть освоение бюджета плюс вовлечение сомнительного персонала.

100% верно. Эти слова да всем чиновникам в уши.

История нескольких #ifdef

КИС, реализующая основные функции автоматизации деятельности торгово-производственной фирмы среднего размера: от бухгалтерии и складов до сборочного производства и сбыта – была

разработана командой из 4–5 человек примерно за полтора года, включая миграцию с предыдущей версии. Система критичная, даже короткий простой оборачивается параличом деятельности фирмы.

Причина столь сжатых сроков? Ясное понимание решаемых прикладных задач, создание соответствующего задаче инструментария, прежде всего, языка бизнес-правил высокого уровня, и подтверждение тезиса Брукса о многократно превосходящей производительности хороших программистов по сравнению с остальными.

Ultima-S – КИС из коробки

Техническая культура – это не производства и знания, а люди, умеющие это делать и применять.

Можно и мартышку научить «мыслить шаблонами», можно и в роботов вложить необходимые конвейерные знания. Но всегда нужны будут те, кто будут дрессировать мартышек, создавать роботов и мыслить «выходя за плоскость».

Два похода к разработке КИС: «от производства» (надо распланировать и оптимизировать имеющиеся ресурсы производства) и «от бухгалтерии» (надо проанализировать хозяйственные операции и на основе полученных данных спрогнозировать дальнейшие действия).

Думаю, здесь автор всё достаточно подробно и чётко изложил.

Проектированием системы на самом деле должен заниматься не технарь, а маркетолог как Стив Джобс. Система – это товар, удовлетворяющий потребности. Маркетолог может понять потребности рынка, сформулировать требования к товару, может оценить осуществимость маркетинговой компании. На деле технический архитектор нужен маркетологу для оценки себестоимости проекта и его сроков, а также для информации о новых перспективных технологиях, тогда маркетолог сможет искать сочетания «потребность + технология».

Всё уже сказано, дополнить мысль нечем.

Однако, тогда мне не понятно за счёт чего первое время держатся стартапы, ведь у них в команде даже толком менеджера нет, я уже и не говорю о маркетологах. Конечно, с одной стороны, я, возможно, слишком идеализирую образ стартапщика, этакого молодца, в одиночку сворачивающего горы с гордо поднятой головой, но, тем не менее, нельзя исключать и такого развития событий, где команда из трёх технарей выстреливает на рынке за счёт пробивной идеи. Наверное, я сам и отвечу на свой сомнительный вопрос: команду, как и проект, в таком случае кто-то более солидный и опытный замечает, а он то, в свою очередь, и подыскивает ребятам, которые не сильны в финансовой грамоте (весьма и весьма условное ограничение) и бизнес-аналитиков, и менеджеров, и маркетологов. А они уже делают своё дело, которому их так же, как и наших умельцев-технарей обучали.

Следуйте совету Джобса: «Обычные художники заимствуют – великие воруют».

Очень и очень спорные слова. Не могу прямо сейчас понять, что же мне в этой фразе не нравится: то ли тот факт, что эти грязные слова были произнесены довольно авторитетным человеком, то ли действительная отрезвляющая правота этих слов, сбивающая мои «розовые очки». Трудно сказать.

Но, стоит заметить, что если эту цитату немного смягчить и развернуть под углом, мол, смотрите на других и берите у них самое лучшее, то я, безусловно, соглашусь. Это даже с психологической стороны полезно. А с технической, на мой взгляд, это одна из вещей, воспитывающая техническую культуру.

Нешаблонное мышление

Выдумывание велосипедов

Зачастую, ещё вчерашние новички, научившись достаточно элементарным вещам, любят порассуждать о том, что изобретение велосипедов – пустое дело. По моим представлениям на воспроизведение большинства из приводимых в книжке «велосипедов» в конкретных случаях у любого специалиста с навыками абстрактного мышления вряд ли должно уходить более часа. Это заметно быстрее изучения самой книги, ее осмысления и, наконец, осознания тех моментов, когда, собственно, надо применить абстрактный слепок в реальной жизни.

Сложнее обстоит дело с теми, кто только начинает свой путь. Прочтение сего труда новичком, как мне кажется, является прямым аналогом попадания в прокрустово ложе диктуемой парадигмы. Потому что книга описывает набор решений, а неокрепшему за недостатком практики уму проектировщика надо научиться самому находить такие решения и пути к ним. Для чего гораздо эффективнее первое время «изобретать велосипеды», нежели сразу смотреть на готовые чужие. На чужие надо смотреть, когда придуман хотя бы один собственный, чтобы понять, насколько он несовершенен, и выяснить, каким же путём можно было бы прийти к лучшим образцам велосипедов данной модели.

Thinking in patterns

Несколько позднее я наткнулся в магазине на книгу с названием, претендующим на звание наиболее абсурдного из встречавшихся. Оно звучало как «Thinking in patterns». В переводе на русский язык – «Мыслить шаблонно».

Ещё недавно привычка шаблонно мыслить считалась в инженерном сообществе признанием ограниченности специалиста, наиболеепригодного для решения типовых задач с 9 утра до 6 вечера. Теперь не стесняются писать целые книжки о том, как научиться шаблонному мышлению…

Думать головой

Наследование = обобщение

Откажитесь от термина «наследование», который искажает смысл действий. Мы обобщаем.

Обобщаемые классы должны иметь сходную основную функциональность.

Основная ошибка – обобщение по неосновному функциональному признаку.

Ошибка приводит к построению иерархии по неосновному признаку. Когда внезапно найдётся ещё один такой признак, возможно, более существенный с точки зрения прикладной задачи, обобщить больше не удастся: придётся ломать иерархию или делать заплату в виде множественного наследования, недоступного во многих объектно-ориентированных языках. Или пользоваться агрегацией.

Не увлекайтесь обобщением. Ошибки тоже обобщаются и уже в прямом смысле этого слова наследуются. Исправление по новому требованию может привести к необходимости сноса старой иерархии, содержащей ошибки.

Эмпирическое правило

Глубина более двух уровней при моделировании объектов предметной области, вероятнее всего, свидетельствует об ошибках проектирования.

Освобождение памяти

Наиболее очевидное преимущество – программисту не надо заботиться об освобождении памяти. Хотя при этом все равно нужно думать об освобождении других ресурсов, но сборщик опускает планку требуемой квалификации и тем самым повышает массовость использования среды. Но за все приходится платить. С практической стороны недостатки сборщика известны, на эту тему сломано много копий и написано статей, поэтому останавливаться на них я не буду. В ряде случаев недостатки являются преимуществами, в других – наоборот.

Черно-белых оценок здесь нет. В конце концов, выбор может лежать и в области психологии: например, я не люблю, когда компьютер пытается управлять, не оставляя разработчику достаточных средств влияния на ход процесса.

Журнал хозяйственных операций

Дам совет начинающим: учите сиквел, транзакции, уровни изоляции, и будет ваша система быстрой и надёжной.

UML и птолемеевские системы

В UML должно настораживать уже самое первое слово – «унифицированный». Не универсальный, то есть пригодный в большинстве случаев, а именно унифицированный, объединённый.

Всё верно. Если абстрагироваться, то можно объединять много разного нехорошего и неполезного. От этого мы результат хороший не получим, сколько бы мы не надеялись на то, что «минус на минус даст плюс» или, что какая-то мелочь, затесавшаяся, не вылезет на поверхность. Является ли подобным объединением UML? Нет, во всяком случае, уверен, есть задачи, когда использование UML облегчает производственный процесс или процесс разработки.

Графические образы – это, конечно, аналог слов, но не всякий набор слов является языком. Поэтому UML до языка ещё далеко. Язык, даже естественный, можно автоматически проверить на формальную правильность. Входящий в UML OCL хоть и является языком, но выполняет лишь малую часть работы по проверке, и не собственно модели, а её элементов.

Очень простой, но верный признак проблемы – отсутствие в инструменте моделирования команды «Проверить». И если условной кнопки «Check model» в панели не предусмотрено, то ваш набор инструментов и методов для проектирования программного обеспечения является просто продвинутым редактором специализированной векторной графики с шаблонами генерации кода.

Итак, 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.

У Киплинга есть замечательный рассказ «Как было написано первое письмо». Про неудачный опыт использования графической нотации первобытными людьми. Но если в рассказе все закончилось хорошо, то в современном проекте была ситуация, когда двум проектировщикам по отдельности поручили делать диаграммы переходов для одного и того же набора классов. Получилось очень «унифицированно»: найти хотя бы одну пару общих элементов в двух диаграммах было затруднительно. Если методология проектирования действительно существует, то работа двух людей даёт сопоставимый результат.

Это [UML] идеальный инструмент для создания птолемеевых систем. Только если модель Птолемея является геоцентрической, то прецеденты использования – модель заказчико-центрическая.

Очень красивое высказывание, надо взять на заметку: «заказчико-центрическая модель».

Слишком часто формулировка «лучше плохой, чем никакой» стала широко применяться, что настораживает.

Поддерживаю.
На днях была опубликована статья в онлайн-журнале TheJournal (ссылка: https://tjournal.ru/p/everything-is-buggy), тема которой, если вкратце, то была такая: багги из приложений и операционных систем никуда не исчезнут, придётся с этим смириться, уважаемые граждане. Посему, вспоминается эта статья при чтении формулировки.

Для тиражируемых продуктов и ориентированного на специализации проектного софтостроения наиболее интересен подход с разработкой собственных языков, где необходимость использования UML неочевидна. Остаются, по большому счёту, заказные проекты без видимых перспектив повторных внедрений, где само слово «моделирование» может вызвать реакцию «пиши код, мы тут проектируем только снизу-вверх».

К великому счастью, программисты системы не занимались тогда «эволюционной» разработкой, а знали область, в которой работают, и добросовестно поддерживали проектную документацию в актуальном состоянии.

Не могу сказать, что сейчас прямо-таки всё-всё-всё не так. Да, то, что раньше было узким, сейчас расширело, набрало массовость, либо исчезло за ненадобностью. Но, тем не менее, необходимость разбираться и делать это самостоятельно никуда не делась. Это одна из первых вещей, на которые смотрят во время испытательных сроков более опытные дяденьки и тётеньки. Да и сами опытные дяденьки и тётеньки на то и гордо носят название «опытные», что знают область не понаслышке, а по работе вживую. А ценнее этого никакая философия быть не может.

Что имеем по итогам более чем 10-летнего развития технологий? Появилась возможность стандартизировать хранение и доступ к большому массиву данных, используя вполне обычное серверное оборудование корпоративного класса и 64-разрядную промышленную СУБД. Возникла необходимость переделывать программное обеспечение из-за утраты поддержки среды разработки поставщиком и соответствующих компетенций на рынке труда.

Но я совсем не уверен, что ещё через 10 лет новые подрядчики, вынужденные в очередной раз переписывать систему, найдут документацию в том же полном виде, в котором нашли её мы. Если вообще найдут хоть что-то, кроме кода, остатков презентаций и наших отчётов.

«Оптисток», или распределённый анализ данных

Наибольшую трудность, кстати, вызывает не сам импорт данных, а поиск в компании людей, которые могут знать, где эти данные взять. Всё-таки 20 тысяч таблиц SAP R3 и примерно столько же в корпоративном хранилище – не шутка, поэтому без хороших проводников раскопки источников информации обернутся поиском иголки в стоге сена.

Архитектура сокрытия проблем

Просто история о том, как «разрабы» скрывали свои огрехи, основанные на пересмотре огрехов других. Не более того.

Code revision, или Коза кричала

Хуже, когда вполне программистский коллектив умудряется годами работать без системы контроля версий исходников, и тогда в коде половину объёма составляют закомментированные куски многолетней давности. Выбросить их жалко, вдруг пригодятся. Но и контроль версий с архивацией не спасает от цифровой пыли десятилетий. В подобных залежах порой можно обнаружить настоящие образцы софтостроительных антипрактик.

Уж увольте, но я не могу понять этого высказывания. Может быть, я в силу наивности так считаю, но, выгляньте в окно, на дворе 2015й год заканчивается, Git с Mercurial были придуманы лет десять назад и получили очень и очень широкое распространение среди программистов разного покроя и пошива. Даже на курсах людей учат пользоваться системами контроля версий.

Намотав на ус всё вышесказанное, взгляните ещё раз бегло на авторский текст. Что это за «вполне программистский коллектив»? Ребятки пишут софт на коленке для шарашкиных конторок? Если только так, ибо компании покрупнее, у кого в штате есть хотя бы один человек, который сечёт в чём-то кроме программирования, смекает, что надо бы как-то эти залежи поудобнее разгребать. Да и стартапщики сейчас не такие уж и забулдыги, чтобы не пользоваться СКВ. Напротив, это модно и приятно пользоваться системами контроля версий.

Насчёт архивации древних записей не могу толком ничего сказать, ибо не сталкивался с этим вплотную. Однако уверен, что гиганты вроде PreForce, Atlassian, BitBucket, GitHub, Mercurial что-то имеют по этому поводу. По крайне мере, это «что-то» достаточно хорошо сделано, ибо на том же GitHub можно найти очень и очень старенькие файлы, даже проектики, созданные в день открытия системы. И само собой, о выбросе старых исходников речь заходит только тогда, когда одна концепция / парадигма / один инструмент сменяются другими.

Ревизия кода, несомненно, весьма полезная процедура, но как минимум при двух условиях:

• эта процедура регулярная и запускается с момента написания самых первых тысяч строк;

• процедуру проводят специалисты, имеющие представление о системе в целом. Потому что отловить бесполезную цепочку условных переходов может и компилятор, а вот как отсутствие контекста транзакции в обработке повлияет на результат, определит только опытный программист.

Полностью согласен с автором. Зелёненьким проггерам или просто талантливым, но неопытным ребяткам доверять топор ревизии в руки нельзя, они начнут смущать тех, кто ещё зелёнее.

Я бы ещё добавил помимо всего прочего с первых строк кода делать прогонку на тестах, чтобы и guideline языка соблюдались, чтобы и функционал был работающий, чтобы вообще ничего и нигде е смущало и не падало. Обычно, на нормальных проектах так и делают.

Большой ошибкой является привлечение к процессу внутреннего тестирования и обеспечения качества посредственных программистов. По его мнению, компетентность специалиста в этом процессе должна быть не ниже архитектора соответствующей подсистемы. Действительно, ведь оба работают примерно на одном уровне, прост один занят анализом, а другой – синтезом.

И не поспоришь!

Наживулька или гибкость?

Очень важно отделить редкую ситуацию «бизнес меняется еженедельно» от гораздо более распространённой «представления команды разработчиков о бизнесе меняются еженедельно». Если вам говорят о якобы часто изменяющихся требованиях, всегда уточняйте, о чём, собственно, идёт речь.

Да.

*** О, да в этой главе рассказывается про спиральную методологию разработки! ***

Ключевой особенностью гибкой методики является наличие мифологического титана – владельца продукта (product owner), который лучше всех знает, что должно получиться в итоге. На самом деле это просто иная формулировка старого правила «кто платит, тот и заказывает музыку». Именно владелец, за рамками собственно гибкого процесса, гением своего разума проводит анализ и функциональное проектирование, подавая команде на вход уже готовые пачки требований. Размер пачки должен укладываться в интеллектуальные и технологические возможности разработчиков, которым предстоит осуществить её реализацию за одну итерацию.

Согласен.

Методология, по сути, направлена на увеличение времени работы с клавиатурой и не располагает к размышлениям. Пиши код! Поэтому для стимуляции персонала процесс окружен религиозной атрибутикой, манипуляциями, иносказаниями и метафорами. С другой стороны, требования к уровню программиста ограничиваются знанием конкретных технологий кодирования, стандартных фреймворков, «умением разбираться в чужом коде» и «умением работать в команде», уже упоминавшимся в словаре для начинающего соискателя. Способность решать олимпиадные задачки здесь от вас не требуется. Скорее, наоборот, будет помехой.

Позитив для заказчика в том, что, 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. После чего он может переформулировать существующие и добавлять новые требования с учётом уже набитых шишек. Тем не менее с ростом сложности системы возрастает и риск увеличения стоимости внесения изменений. И если проект выходит за рамки бюджета, то «козлом отпущения» становится именно владелец продукта.

Другой позитив для компании-заказчика состоит в непосредственной близости выполняемой подрядчиком работы. Зачастую «гибкие» команды работают на площадке компании и доступны в любой рабочий момент. Если заинтересованному лицу не хватает информации, он может просто подойти и посмотреть на месте, поговорить с исполнителем и тем самым восстановить расстроившееся было душевное равновесие.

Комбо «согласен» x3.

Позитив для подрядчика состоит в том, что «гибкая» разработка позволяет вовлечь в проект как можно больше разработчиков с менее высокими требованиями к квалификации. Это позволяет содержать больше сотрудников в штате, включая оффшорные команды. Будучи поставленным перед выбором между небольшой программистской фирмой с квалифицированным персоналом и софтверхаузом-«тысячником», крупный заказчик в общем случае склонится ко второму варианту.

Логично. За счёт гибких методологий и живут компании. А вы как думали, верили в добрый исход и победу на рынке узкоспециализированных команд?

Заказчик осознаёт, что реализовать спецификации своими силами невозможно, прежде всего потому, что при таком объёме они тем не менее неполные и неизбежно содержат противоречия. Подрядчику же в принципе наплевать на спецификации, он будет крутить итерации, честно реализуя заявленный функционал и отрабатывая бюджет. Получился коровник на подпорках с покосившимися заборами и дырявой крышей вместо современного агрокомплекса? Извините, всё по спецификации, каждые две-три недели вы видели расцвеченные фотографии разных участков возводимого сооружения.

Без комментариев.

Тесты и практика продуктового софтостроения

При изменении спецификаций затраты времени на приведение в актуальные состояния тестов могут быть куда больше, чем на собственно код. Скажем, если пользователи захотят поменять синтаксис SQL в СУБД и писать SEARCH вместо SELECT, то это одно изменение продукта в одном месте приведёт к переписыванию почти всех тестов. Если для СУБД такие пожелания пользователей – редкость, то для менее стандартных программ – обычное дело.

Ууу, батенька, да если так пошло, то это просто диагноз. Давеча беседовал со своим куратором на работе, надо было foreign key-ю атрибутику поменять, чтобы шло удаление. Так вот задал я ему вопрос, как мне всё-таки это дело организовать? В миграции для БД? На что он мне взял и заявил резко, но правильно: нет и ещё раз нет, исключительно в логике программы. Почему? Да потому что начнут ребят использовать другой сервер, или, вот, пользователь захочет заниматься такой чушью, то днями и ночами придётся весь маппинг к БД переписывать и перестраивать. Кому это надо?!

Так и тут. Тесты должны проверять логику программы, а не формализмы разные, вроде словечек SELECT & SEARCH. Да, это понадобится, если мы хотим оптимизировать что-то, но это уже отдельный разговор, я б даже сказал, постскрам какой-то. А на производствах до этого постркрама ещё доползти целым надо…

Сбои, которые могут выловить тесты (повторяемый сбой в модуле, ранее уже исправлявшийся), – довольно редкое дело. Гораздо чаще сбои возникают в интеграции разных технологий, которые сложно автоматически протестировать. Например, при работе под таким-то браузером при таких-то настройках вот этот JavaScript работает неправильно.

Редкое, однако, это зависит от платформы и развитости подходов к тестированию на ней. По поводу интеграций – согласен. Но всё это запросто можно отследить и протестировать, QA умеют.

Наличие модульных тестов сильно затрудняет масштабный рефакторинг.

Да оно даже микро рефакторинг усложняет.

Одни из наших конкурентов широко используют agile-методы и TDD, но что-то оно им не очень помогает писать безошибочный код. Мы сравнивали количество найденных проблем в течение месяца-двух после major release, у нас показатели лучше в разы, если не на порядок. Частый выпуск версий просто не позволяет им довести код до ума и провоцирует исправление старых и серьёзных проблем методом написания «залепени».

Если мы выпускаем продукт раз в 2 недели и находится серьёзная ошибка в версии годичной давности, то нам её нужно будет исправить примерно в 40 ветках. Конечно, править 40 веток кода никто не будет, пользователям сообщат, что ошибка будет исправлена в следующей версии, многие пользователи перейти на неё не смогут к большой радости конкурентов с предложениями типа competitive upgrade offer.

Верно.

Излишне религиозная атмосфера превратила вполне здравую и работающую с 1970-х годов технологию модульных тестов в настоящий карго-культ. Адепты 100 % покрытия кода модульными тестами считают, что это обеспечит успех проекту.

Верю, видел, знаю. Гиблое дело. Но тут такая вещь: балансировать в этом быстром мире, да ещё и со скрамами и гибкими методологиями едва получается, кренит то в одну сторону, то в другую.

Понятия не имею почему, но сейчас, я представил себе it-сферу этакой мамашкой, у которой миллионы раскрытых ртов, которых надо кормить, причём кормёжка - это отбор еды у других детёнышей. Так ладно, тут есть эти, вокруг ещё больше рождается, а мамашка не справляется и начинает тихонечко сходить с ума. А много детёнышей появляется, потому что о мамане много всего такого хорошего говорят, все вокруг её расхваливают. Ну и от мужиков-отраслей отпора нет. И получается, наверное, что от этого мамашка, которая была более-менее сконцентрирована в себе, в университетах и философско-математических концепциях, решила пошалить. Отчего и стала вести себя, как проститутка – прыгать на каждого и рекламировать себя не лучшей стороны, забыв о том, какой была изначально.

Печально это, ой печально. Как из этого оврага выезжать все вместе будем? Не знаю, наверное, только взрослеть, взрослеть…

В завершение темы было бы непростительно не вспомнить о производственной системе Toyota, которую почему-то считают основой того же скрама. Ключевой особенностью системы в Тойоте является принцип «дзидока» (jidoka), означающий самостоятельность людей в управлении автоматизированной производственной линией. Если рабочий видит нарушение качества продукции или хода процесса, он имеет право, повернув соответствующий рубильник, остановить всю линию до установления причин дефектов и их устранения.

Внедрение таких методик вполне возможно и вне рамок японского общества. New United Motor Manufacturing Inc (NUMMI) – знаменитое совместное предприятие Toyota и General Motors, ещё в 1970-х годах вошло в «кейсы» бизнес-школ как пример повышения эффективности и качества через смену культуры работы.

Качество программного продукта – многозначное и сложное понятие. Производственная культура – ещё более сложное. В одном можно быть уверенным: ни о какой культуре софтостроения не может идти и речи, если любой программист из коллектива не способен остановить бессмысленный циклический процесс для выяснения, какого же рожна по историям заказчика потребовалось обобщать четырёхногих коров и обеденные столы.

Как вы заметили, я стараюсь перемежать темы, несколько напрягающие мыслительные органы, с занимательными зарисовками из жизни. Продолжим традицию историей о совместимости компонентов сложной тиражируемой системы.

Заметили, это точно. Пару конспектов назад уж как заметили. Мне кажется, что гораздо полезнее для нас, интересующихся этой темой, было бы не читать разглагольствования автора и искать там суть, а

просто с ним пообщаться и таким образом получить необходимый опыт. А то получается так: мы пишем конспект по материалу, который весьма вторичен, часть его отсырела, часть его вообще непонятна. То, что осталось, носит свою специфику, которую очень трудно обобщить, тем более нам, «зеленоватым студентам».

Посему, считаю, что смысла в частых заметках, взятых из этого материала, нет. Ограничимся мелкими. Но ёмкими.

Итог главы:

Какие же всё-таки Microsoft потешные. И не они одни такие, просто такого рода «фейлы» (fails) прочно ассоциируются именно с ними, с их продукцией. И, тем не менее, их продукция остаётся каким-то призрачным гарантом качества, благо, дизайн уже медленно подыхает.

А вообще, для меня мораль сей басни была такой: корпоративное решение другому корп. решению рознь. А мелкософты продолжают мутить воду, от которой страдают и разрботчики, и весь проект в целом (от простоя).

Возможно, что я не увидел чего-то ещё, что хотел вложить в рассказ автор. Что ж, жаль.

Программная фабрика: дайте мне модель, и я сдвину Землю

В управляемой моделями разработке и в программной фабрике наиболее интересной возможностью является генерация кода, скомпилировав который, можно сразу получить работающее приложение или его компоненты. Мы проектируем и сразу получаем нечто работающее, пусть даже на уровне прототипа. Уточняя модели, мы на каждом шаге имеем возможность видеть изменения в системе. Проектирование становится живым процессом без отрыва от разработки.

А вот это интересный и полезный подход. Во многом, он зависит от платформы, на которой выстраивается взаимодействие с базой, моделью и так далее. Если платформа очень перенасыщена деталями, то от проектирования она будет отвлекать на хотфиксы и описанные ранее консультации с ребятками из microsoft. Иначе можно спроектировать не совсем гибкий по функционалу слепок будущей системы, которую можно уже переносить со всеми оговорками на более изощрённую платформу. К примеру, начать разработку на RubyOnRails и перенести всё на Java.

В качестве решения перечисленных проблем появились так называемые двусторонние CASE-инструменты (two way tools), позволяющие редактировать как модель, непосредственно видя изменения в коде, так и, наоборот, менять код с полу– или полностью автоматической синхронизацией модели. Зачастую, такой инструмент был интегрирован прямо в среду разработки.

Вот к таким вещам у меня двоякое отношение, тем более, зная раннюю позицию автора. Да, автоматизировать процесс разработки круто, как говорят, «базару нет», но, не стоит забывать, что смысл работы программиста не в том, чтобы обойтись нажатием 2-3 кнопок и получить готовый результат. В первую очередь, программисты, это люди, которые шевелят мозгами, ищут и исправляют ошибки, радуются написанию хорошего кода и постоянно совершенствуют свой стиль. О каком тогда совершенствовании может идти речь в таком гиперболизированном случае, когда автоматизирован весь процесс разработки? Единственное, это создавать автоматы. Сначала для программ / продуктов, а затем и для создания самих автоматов. Вот вам и рекурсия, круг замкнулся. Хорошо это или плохо? С моей точки зрения, скорее хорошо, ибо придётся напрягать мозги ради изобретения автоматов и совершенствования изобретённых, но всё равно, область применения ума может заметно сузиться, а это огроменный минус.

Надеюсь, такого никогда не произойдёт, хотя бы потому, что автоматизировать всё нельзя.

Лень – двигатель прогресса, особенно когда надоедает переписывать генераторы кода и подстраивать относительно стандартные модели под частные требования.

Ой, прямо в точку! Ради этого, собственно и нужна автоматизация. Кому нравится создавать кубики с полного нуля для того, чтобы построить небоскрёб? Чёрт возьми, да кто-нибудь, занимаясь веб-разработкой, начинает с реализации сетевых протоколов и описания структур в плюсах через код ассемблера? Знать, как устроены кубики необходимо, но очень вредно собирать то, что уже давно есть и широко используется, с абсолютного нуля на реальном производстве. ~~Жопа горит.~~ Сроки горят. Да, это необходимый этап, если ты создаёшь что-то инновационное или же учишься (вспомнить хотя бы УП). Но иначе это просто трата времени. Надо просто уметь разбираться в документации так, чтобы БЫСТРО найти необходимое и это применить.

Лампа, полная джиннов

Метафора системы достаточно проста: хочешь генерировать код компонента или слоя – попроси об этом соответствующего «джинна» в форме стандартного «заклинания».

Верно. А тут уж раздолье для всяких мнемонических приёмчиков, которые зависят от платформы и фреймворка.

Язык создан на основе XML, поэтому делать описания можно непосредственно руками в обычном текстовом редакторе.

XML – прошлый век. JSON лучше хотя бы потому, что его легче парсить в какой-нибудь словарь / хэш, который потом можно отдать на растерзание методу из ORM. А XML парсинг… я помню, мы его писал в рамках учёбы. Это была жесть. Плюс, XML затрудняет чтение, на мой взгляд.

Приведу пример описания из рабочего проекта, содержащего один пользовательский тип, один перечисляемый тип, две сущности и одну связь (отношение) между ними.

Это ужас. А где же компактность?! Может быть, я привык к YAML формату из Ruby On Rails, но XML это какое-то кощунство…

Заключительная часть конфигурации представляет собой описания шаблонов.

А вот это хорошо, если ориентироваться и знать. Это дополнительная гибкость, а это плюс.

Слой хранения (СУБД)

Слой домена (NHibernate)

Слой веб-служб и интерфейсов доступа (ServiceStack)

Программа-клиент

Остановиться и оглянуться

Если рассмотреть метрики относительно небольшого проекта, то 40 прикладных сущностей в модели, состоящей примерно из 600 строк XML-описаний, порождают:

- **около 3 тысяч строк SQL-скриптов для каждой из целевых СУБД;**
- **порядка 10 тысяч строк домена;**
- **1200 строк XML для проекций классов на реляционные структуры (таблицы);**
- **около 17 тысяч строк веб-служб и интерфейсов.**

Таким образом, соотношение числа строк мета-кода описания модели к коду его реализации на конкретных архитектурах и платформах составляет около 600 к 30 тысячам или 1 к 50.

Это означает, что оснащённый средствами автоматизации программист с навыками моделирования на этапе разработки рутинного и специфичного для платформ/архитектур кода производителен примерно так же, как и его 50 коллег, не владеющих технологией генерации кода по моделям.

Любое внесение изменений в модель тут же приводит в соответствие все генерируемые слои системы, что ещё более увеличивает разрыв по сравнению с ручными модификациями. Наконец, для генерируемого кода не нужны тесты. Производительность возрастает ещё как минимум вдвое.

Даже если принять во внимание, что доля рутинного и прочего инфраструктурного кода по отношению к прикладному, то есть решающему собственно задачи конечных пользователей, снижается с масштабом системы, есть о чём поразмыслить в спокойной обстановке.

Я уже писал ранее, что я думаю по поводу автоматизации. Хорошая штука, но не стоит ей увлекаться. Да, здорово иметь под рукой инструменты, которые позволяют быстро соорудить каркас обычного приложения (без всяких наворотов, чисто модели, контроллер, вьюхи – в зависимости от используемого паттерна своё наполнение) и работать с порождённым шаблоном, подстраивая его под себя. НО надо не ограничиваться одним шаблоном, смотреть шире и знать, то кроме MVC есть хотя бы MVVM и, соответственно, знать, как ручками, в случае высадки на местность, кишащую аборигенами, написать приложение с выбранным паттерном, чтобы спастись.

Cherchez le bug, или Программирование по-французски

Хаос наступает внезапно

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». Это ещё далеко не полный хаос, но уже давно и далеко не порядок. Все попытки как-то организовать процесс разработки программ, всяческие методологии, применение парадигмы конвейера, стандарты и административные меры худо-бедно, но помогают оттянуть этот критический порог на некоторое время. В идеале – до того момента, когда развитие системы останавливается и она, побыв некоторое время в стабильном состоянии, потихоньку умирает.

Согласен. Добавить нечего.

Одна из проблем организации промышленного производства программного обеспечения состоит в отсутствии каких-либо формальных описаний деятельности программиста. Можно определить в технологической карте, как работает сварщик или каменщик, но как пишет программу программист, зачастую, не знает и он сам. До художника, конечно, далеко не все дотягивают, а вот с деятельностью рядового журналиста «независимой» газеты непосвящённому в софтостроение человеку сравнивать вполне можно. Этакий ядрёный сплав ремесла, некого богемного искусства, со вкраплениями науки, вперемешку с халтурой, шабашкой и постоянным авралом. Попытки же принудить программиста делать однотипные операции противоречат самой цели существования программного обеспечения как самого гибкого из существующих средств автоматизации рутинных процессов и потому изначально обречены на неудачу.

Кстати, если программист говорит вам, что в данном месте программа «должна работать», это значит, что с очень большой вероятностью она не работает, а он её просто не проверял в этом месте. Если же программист уже после обнаружения ошибки говорит: «А у меня она в этом месте работает…», лучше сразу его уволить, чтобы не мучился.

Что-то с памятью моей стало

На самом деле я очень не люблю, когда в комнате, где люди работают большее время дня в сидячем положении и относительной тишине, начинает кто-то мельтешить перед глазами и шуметь. Думаю, многие будут со мной солидарны. Отвлекает.

Три дня в IBM

Консультант, дядька лет пятидесяти, исправил нам эти параметры и пожелал дальнейших успехов. Кстати, у них там вся команда консультантов и менеджеров состояла из мужчин предпенсионного возраста – факт отрадный, если вспомнить откровенную и циничную дискриминацию по возрасту при приёме на работу во многих российских фирмах.

Вот приятно это читать, ибо не понаслышке знаю, каково это. Мой отец, по профессии инженер-конструктор-радиотехник, около 50 своих лет тоже пытался устроиться в наш ПВТ, парк высоких технологий. И вы знаете, что ему сказали, несмотря на то, что за спиной у отца 5 патентов и сотни изобретений? «Вы нам не подходите по возрасту». И ладно он шёл бы обычным рядовым программистом, код клепать. Нет, это работа для таких зелёных, как я сам, студентов. Мой отец брал рамку выше – тимлид, менеджер, юнит-менеджер. Отец и на ASMe писал, и на PHP с C++, и веб-технологиями занимается даже сейчас, причём каждый день на этом зарабатывает. Но, увы, на все его заявки в ПВТ ответили циничным отказом. Из-за чего я не могу воспринимать ПВТ в нашей стране должным образом, по крайне мере, без взаимного цинизма.

Удалось, буквально ткнув пальцем в небо, выяснить причину ошибки в модуле. Ею оказался настолько искусно криво написанный цикл в одной из многочисленных функций, что не сразу была видна его нехорошая тенденция при определённых условиях превращаться в бесконечный.

Вспоминается Константин Антонович Зубович и его заветы «не использовать цикл for». ☺

Хорошо там, где нас нет

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…

Я нахожу это странным. Очень. Да, конечно, у стартапа главное это реализованная идея, но… как же формализованная чистота и прагматичность, прущая изо всех щелей? Как-то это непривычно, что ли. И мне кажется, что скорее именно мелкие компашки в первую очередь работают с системами управления версиями и следят за дубликатами таблиц, ибо если у них там будут проблемы, то из мелкой компании они ни во что не превратятся.

О технических книгах

Дефрагментация мозгов

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».

Но в то же время, плоды науки нужны всем. А направлять в науку никто не хочет. Наоборот, выгодно рынку делать людей бестолковыми.

Нынешняя «гуглизация» позволяет быстро находить недостающие фрагментарные знания, зачастую забываемые уже на следующий день, если не час. Поэтому ценность книг как систематизированного источника информации, казалось бы, должна только возрастать.

Кроме стоящих вне конкуренции учебников остаётся в основном только конкретизация – узкоспециализированная проработанная технологическая тема либо обобщение концептуальных наработок и практического опыта.

Долгий опыт университетского преподавателя Павла Андреевича Калугина ныне обогатился добавлением в курс информатики такой эклектики, как Enterprise Java. Проблемы подобные курсы вызывают больше не у студентов, а у обладающего системным образованием и мышлением преподавателя, вынужденного каким-то образом связывать противоречивое и выстраивать логику там, где её изначально не было.

Почему же относительно легко студентам? Дело, прежде всего, в смене образа мыслей, а может быть, и восприятия самой действительности. Мир – это такая очень большая и сложная компьютерная игра, а преподаватели, соответственно, должны обучать не премудростям стратегий познания мира, а практическим приёмам, секретным кодам и даже шулерству, чтобы пройти в этой игре на следующий уровень. Этакие «мастера ключей» из Матрицы. Про то, что в игре продукты на столе появляются прямо из холодильника, тоже стоит упомянуть.

Да, за собой тоже такое замечаю. Но тут это самое шулерство во многом преподносится с красотой, отчего на душе становится благодатно и хорошо.

Много копий сломано в обывательских дискуссиях о так называемом клиповом мышлении, шаблонности, «плохом образовании» вкупе с апокалиптическими прогнозами и прочим. Хотя на самом деле пока непонятно, к чему приведёт тенденция в перспективе ближайших десятилетий. Во фрагментарном «клиповом» мышлении есть и свои плюсы: способность быстро решать типовые задачи, широкая квалификация и мобильность, меньшие затраты на массовое образование. Минусы тоже ясны. В апокалипсис технократии я не верю, всегда будут рождаться способные дети и существовать ограниченное число учебных заведений, дающих жутко дорогое фундаментальное образование, вроде того, что было общедоступным в СССР. Видимо, этого должно хватить на поддержку критичных технологий цивилизации и дальнейшее их развитие.

Является ли фрагментарное мышление приспосабливанием к многократно увеличившемуся потоку информации? Отчасти да, только это скорее не адаптация, а инстинктивная защита. Чтобы осознанно фильтровать информацию о некоторой системе с минимальными рисками пропустить важные сведения, нужно иметь чётко сформированные представления о ней, её структуре и принципах функционирования. Если, например, у стажера нет знаний о СУБД в целом, то курс по SQL Server – конкретной её реализации, сводится к запоминанию типовых ситуаций и решений из набора сопутствующих практических работ. Вся теоретическая часть при этом просто не проходит фильтры.

В такой ситуации альтернативой ухода от информационных потоков и некачественного образования во фрагментарную реальность является самообразование на базе полезных книг. Потому что хорошая книга – самый эффективный способ дефрагментации ваших мозгов.

Полностью поддерживаю. В особенности относительно самообразования. Считаю, что каждый должен им заниматься. В каждого это надо вложить. Иначе человечество просто погибнет от жажды наживы и развлечений.

Простые правила чтения специальной литературы

Итак, настоящий исследователь должен:

• быть достаточно ленивым. Чтобы не делать лишнего, не ковыряться в мелочах;

• поменьше читать. Те, кто много читает, отвыкают самостоятельно мыслить;

• быть непоследовательным, чтобы, не упуская цели, интересоваться и замечать побочные эффекты.

Когда натыкаешься в Сети на очередное несдержанное восклицание «must have», подобная постановка вопроса коробит сама по себе. Представьте, что кто-то открыл для себя и полюбил варёный лук. И теперь если ты его не «must have», то как бы и не в струе, и вообще от жизни отстал. Смешно, конечно, но шутки шутками…

А такое вообще везде, развелось много таких крикунов-максималистов.

Если, пролистав с полсотни страниц, вы не сделаете ни одной пометки, то можете смело закрывать сие произведение. В топку бросать, наверное, не надо, но поберегите своё время. Если же к концу прочтения вам хватит листа А4, исписанного убористым почерком, для всех пометок, то можете считать, что чтение прошло не зря. Ну а если одного листа не хватило… Тогда смело идите на свой любимый интернет-форум и там поделитесь с коллегами своими находками. Когда вместо малоинформативного «must have», используя все тот же испещрённый пометками лист, вы сможете передать пользу от прочитанного, коллеги будут вам чрезвычайно признательны за рекомендации и собственное сэкономленное время.

Здорово сказано.

На этом и закончим.

Григорьев Антон

Часть 1. О нашей профессии

Раньше программисты были привязаны к ЭВМ, теперь в большинстве случаев программист может не принимать во внимание особенности тех устройств, на которых будет выполняться их программа.

Программистом следует называть специалиста, способного как самостоятельно формализовать задачу (привести её к виду, пригодному для решения на компьютере), так и воспользоваться стандартными средствами её решения на ЭВМ.

Учёный и классик жанра научной фантастики Иван Ефремов писал: «Красота – это высшая степень целесообразности в природе, степень гармонического соответствия и сочетания противоречивых элементов во всяком устройстве, во всякой вещи и во всяком организме».

Нельзя «сделать красиво», если рассматривать софтостроение лишь как искусство и средство самовыражения. Любить себя в софтостроении, а не софтостроение в себе. Тогда красота рискует так и остаться не воплощёнными в жизнь эскизами. Невозможно обойтись без знаний технологий производства и хороших ремесленных навыков.

Программирование нельзя целиком причислить ни к искусству, ни к ремеслу, ни к науке, ибо всё в нём должно быть гармонично.

Имея возможность потреблять, не отдавая взамен свой труд, останетесь ли вы профессионалом в своей сфере?

В данный момент наблюдается сильный перевес между производством софта и рынка услуг, связанного с ним, в пользу рынка услуг (10% к 90%) по оптимистическим подсчётам.

Отсюда неутешительный вывод для писавших программы в школьных кружках: количество проектов, где потребуется ваша квалификация, намного меньше количества некритичных заказов, а большинство ваших попыток проявить свои знания и умения столкнется с нелояльной конкуренцией со стороны вчерашних выпускников курсов профессиональной переориентации. На практике это означает, что вам, возможно, придётся снижать цену своего труда и готовиться к менее квалифицированной работе.

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

Как вы помните, софтостроение на 90 % находится в сфере услуг. Если вы не работаете на производстве у одного из поставщиков тиражируемого программного обеспечения, то взаимодействия типа «человек –человек» становятся необходимым и важным элементом повседневной работы.

В постиндустриальной экономике сфера услуг занимает более 50 % деятельности, и эта доля растёт, например, в США она уже близка к 70 %. Представьте себе ваш бывший школьный класс, где к работе в обслуживании ориентированы не более 25 %. Откуда же брать недостающих, да при этом еще и услужливых? Проблема, удовлетворительных решений которой на сегодняшний день не найдено. Поэтому и обсуждают введение пособий типа БОД: пусть лучше получают свой минимум и занимаются чем хотят, чем портят отношения с клиентами, мешая производительному труду остальных.

Не надо путать мотивацию и стимуляцию. Мотивация – исключительно внутренний механизм. Чтобы управлять им, необходимо залезать в этот самый механизм, в психику. Напротив, стимуляция – это внешний механизм. Он основан на выявлении мотивов и последующем их поощрении или подавлении. Управлять мотивацией, то есть целенаправленно изменять психологию и выстраивать набор стимулов – это «две большие разницы». Первое, по сути, требует изменения самих людей,

второе – это использование имеющихся у них мотивов. Мотивировать же можно только свои поступки, но никак не образ действия окружающих.

Мнение

Сложно сказать, что наиболее полезным из прочитанного мною. Было безусловно интересно узнать про истинную ситуацию IT-сфере. Про то, что дела обстоят вовсе не так радужно и романтично, как может казаться многим (например, мне). Но, думаю, не стоит принимать этот текст за подробную аналитику проблемы и чёткий план к действию. Скорее цель автора – побудить к размышлениям и изысканиям.

Очень радует, что уже в первой части книги даются некоторые практические советы и ведётся разговор не только о техническом аспекте вопроса (я имею ввиду упоминания о творческой подоплёке и о мотивации).

Таким образом, мне кажется, эта книга может помочь в своего рода осознании себя и своего места в качестве софтостроителя (как я и говорил, на определённые рассуждения она наталкивает).

Часть 2. Программирование и аппаратура

Доступность информации снизила ее значимость, ценность стали представлять не сами сведения из статей энциклопедии, а владение технологиями.

Технологии в аппаратном обеспечении, «железе», подчинены законам физики, что делает их развитие предсказуемым с достаточной долей достоверности. Зная, какие работы ведутся в лабораториях, можно предугадывать потолок их развития и предполагать сроки готовности к практическому использованию.

В противоположность этому, мир программных технологий основан на математических и лингвистических моделях и подчинён законам ведения бизнеса. Крупные капиталовложения, сделанные в существующие средства разработки, инфраструктуру и обучение пользователей, должны окупаться независимо от значения синуса в военное время, релятивистских поправок и элементной базы ЭВМ. Вывод: радикальных изменений в софтостроительной сфере ожидать не следует, ситуация находится под чутким контролем крупных корпораций и развивается эволюционно.

В софтостроении нет такой стандартизированности как в проектировании аппаратуры. В софтостроении нет стандартных компонентов, лишь достаточно сложные подсистемы. Возможность собирать изделия из «кубиков» стала предметом зависти софтостроителей, вылившейся в итоге в компонентный подход к разработке.

То есть софт в общем более сложен для создания в плане его непредсказуемости.

Программы невозможно разрабатывать как аппаратуру из-за ряда ограничений. И, по-видимому, в их основе лежит независимость программ от каких-либо точных правил (законов физики, например).

Безысходное программирование – это программирование без «исходников». То есть мы пишем свой код, не имея исходных текстов используемой подпрограммы, класса, компонента и т. п. Когда необходимо обеспечить гарантированную работу приложения, включающего в себя сторонние библиотеки или компоненты, то, не имея доступа к их исходному коду, вы остаётесь один на один с «чёрным ящиком».

При этом ответственность за конечный продукт и ошибки в нём лежит на вас.

Знаю ли чем выгоден ООП и как вообще его применять?

Мнение

Опять-таки было озвучено много тем, о которых стоит задуматься. И не согласиться с чем-то не получится, ибо все эти в новинку для меня и какой-то целостной картины в голове нет.

Поднятые темы про эффективность разработки, про профессионализм в плане применимости технологий и идей, сравнение подходов к разработке софта и аппаратуры – всё это безусловно занятные темы, и они заставляют поразмыслить о своей компетентности, о своём профессионализме. То есть мало того, что такие источники являются пищей для размышлений, они в довесок могут стать своего рода стимулом к росту, к развитию.

Часть 3. Технологии

Давайте вспомним историю. Успех в 1994–1995 годах первой версии бесплатной и открытой платформы PHP, называвшейся тогда Personal Home Page, показал, что веб быстрым темпом трансформируется из источника статической информации в среду динамических интерактивных приложений, доступных через «стандартный» проводник-браузер.

Странно осознавать, что когда-то создание динамических веб-страниц было прорывом.

Ниже я объясню, почему взял слово «стандартный» в кавычки. Microsoft не могла остаться в стороне и выдала собственное решение под названием ASP (Active Server Pages), работающее, разумеется, только под Windows. 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. Вскоре выяснилось, что привыкшему к интерактивности полноценных приложений пользователю одних лишь кнопок не хватает. Тогда и в браузеры (то есть на стороне клиента) тоже включили поддержку скриптовых языков. В итоге исходная веб-страница, ранее содержавшая только разметку гипертекста, стала включать в себя скрипты для выполнения вначале на сервере, а затем и на клиенте. Можете представить, какова была эта «лапша» на сколько-нибудь сложной странице ASP. Многие сотни строк каши из HTML, VBScript и клиентского JavaScript.

Последующая эволюция технологии была посвящена борьбе с этой лапшой, чтобы программный код мог развиваться и поддерживаться в большем объёме и не только его непосредственными авторами. На другом фронте бои шли за отделение данных от их представления на страницах, чтобы красивую обёртку рисовали профессиональные дизайнеры-графики, не являющиеся программистами. Однако, несмотря на значительный прогресс за последние 15 лет, производительность разработки пользовательского интерфейса для веб-приложений в разы отстаёт от автономных приложений, тех самых, что «компонентокидатели» на Visual Basic, Delphi или C++ Builder делали 15 лет назад.

Действительно, переносить автономное приложение между разными операционными системами и аппаратными платформами трудно. Поэтому на первый взгляд идея универсального программируемого терминала, которым является веб-браузер, поддерживающий стандарты взаимодействия с веб-сервером, выглядит привлекательно. Никакого развёртывания, никакого администрирования на рабочем месте. Именно этот аргумент и стал решающим в конце 1990-х годов для внедрения веб в корпоративную среду. Гладко было на бумаге, но забыли про овраги…

Но выяснилось, что есть большие сложности. А капиталовложения уже были произведены, поэтому отступать было поздно.

Поэтому при разработке веб-приложений достаточно было согласовать внутренние требования предприятия с возможностями разработчиков. К началу 2000-х годов установился фактический стандарт корпоративного веб-приложения: Internet Explorer 6 с последним пакетом обновления под Windows 2000 или Windows XP. Под эти требования за 10 лет было написано великое множество приложений. А когда пришла пора обновлять браузеры, внезапно выяснилось, что их новые версии далеко не всегда совместимы с находящимися в эксплуатации системами. И по этой причине простое обновление Internet Explorer 6 на 7 вызовет паралич информационных систем предприятия.

Напоследок хочется пожелать коллегам, ответственным за выбор технологий, всячески обосновывать необходимость использования веб-интерфейса в вашей системе, принимая в рассмотрение другие пути.

Появившись в 1995 году, технология Java сразу пошла на штурм рабочих мест и персональных компьютеров пользователей в локальных и глобальных сетях. Наступление проводилось в двух направлениях: полноценные «настольные» (desktop) приложения и так называемые апплеты24, то есть приложения, имеющие ограничения среды исполнения типа «песочница» (sandbox). Например, апплет не мог обращаться к дискам компьютера.

В 2007 году Sun утверждала, что среда исполнения Java установлена на 700 миллионах персональных компьютеров25, правда не уточнялась её версия. В декабре 2011 года уже новый владелец – корпорация Oracle – привёл данные о том, что Java установлена на 850 миллионах персональных компьютеров и миллиардах устройств в мире26. Но поезд ушёл, развитие приложений на десктопах сместилось далеко в сторону по пути начинённых скриптами веб-браузеров, а рост количества мобильных устройств положил конец монополии «персоналок» в роли основного пользовательского терминала.

Итак, итог к 2012 году. Во-первых, «старые» технологии вроде автономного оконного кроссплатформенного приложения на Lazarus/FreePascal, Delphi XE или Qt/C++ по-прежнему позволяют сделать то, что нельзя сделать «новыми и прогрессивными». Во-вторых, ценность Silverlight по сравнению с полноценным. NET на уровне развёртывания практически нулевая. Видимо, по этой причине Microsoft недавно закрыла веб-сайт silverlight.net, в очередной раз оставив разработчиков в интересном положении. Из продвигаемых Microsoft за последние 10 лет технологий для разработки полноценных пользовательских интерфейсов, не заброшенных на пыльный чердак, остался только WPF, имеющий весьма сомнительную ценность для небольших коллективов и отдельных разработчиков. WPF – это ниша крупных автономных Windows-приложений. Кроме того, сама по себе она невелика, в ней уже есть WinForms – более простой и быстрый в разработке фреймворк, к тому же переносимый под Linux/Mono. Поэтому при соответствующих ограничениях развёртывания выбор попрежнему лежит между веб-браузером или условным Delphi, хочешь ты этого или нет…

Мнение

На самом деле я так и не понял, о чём хотел сказать автор в главе «Карманные монстры». Разработка неоправданно усложняется за счёт недостаточной гибкости и квалифицированности разработчиков и менеджеров? Я имею явно меньше опыта в реально разработке программного обеспечения, чем автор. И, возможно мне повезло, но в моём случае разработка проекта если и усложнялась (по настоянию так сказать менеджера), то только ради повышения логичности и структурности проекта и кода. То есть, всегда очень легко быстро написать код как ты его понимаешь и как он сразу пришёл к тебе в голову. Так называемый метод «Лишь бы работало». Но стоит немного призадуматься о качестве своего кода, об его эффективности. И вообще о том, что с ним будешь работать не только ты сам. В таком случае можно (и нужно) потратить какое-то время на структурирование кода, на «обмозгование» лучших путей. Лично для меня сложно расчертить грань между необходимым и излишним усложнением. Наверное, это дело опыта.

Было интересно узнать про историю веб-сферы. Сам я web-разработкой не интересуюсь, и не предполагал, что это было настолько востребовано и перспективно (хотя сейчас это безусловно так). Также не предполагал, что сейчас с этой сферой всё так плохо, как описано в книгах. Хотя я, как банальный потребитель, скорее всего пользуюсь незначительно малой частью возможностей моего браузера. Так что с подобными проблемами я никогда не сталкивался. Хотя представляю, какой головной болью такая ситуация оборачивается для профессиональных веб-разработчиков. И недоумеваю, как такую ситуацию допустили. То есть в сообществе разработчиков всё, как и в обществе людей целиком: человеческая масса просто куда-то движется, пытаясь развиваться, при этом допуская

ошибки, которых можно было избежать, и не задумываясь о последствиях. То есть никто ничем не управляет (просто сумбурная деятельность отдельных личностей и коллективов), и всё идёт своим чередом. В итоге имеем то, что имеем.

Автор описывает тернистый путь технологий для создания пользовательских графических интерфейсов. И тут пару вещей мне непонятна. Почему автор описал Java как совсем провалившеюся технологию? Разве не является Java сейчас одним из самых востребованных языков? Разве не имеет она множество средств для создания графических интерфейсов (JavaFX, Swing), в том числе и в веб-среде? Конечно я могу просто не понимать того, что именно пытается донести автор (в силу малого опыта и знаний). Либо автор чего-то недоговаривает.

Часть 4. ООП и ORM

ООП

В начале широкой популяризации ООП, происходившей в основном за счёт языка C++, одним из главных доводов был следующий: «ООП позволяет увеличить количество кода, которое может написать и сопровождать один среднестатистический программист». Приводились даже цифры, что-то около 15 тысяч строк в процедурно-модульном стиле и порядка 25 тысяч строк на C++.

Собственно, и Бьёрн Страуструп, создатель C++, прежде всего преследовал цели увеличения производительности своего программистского труда.

Термин «Ад Паттернов» может показаться вам незнакомым, поэтому я расшифрую подробнее это широко распространившееся явление:

- 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 сложных и многоуровневых конструкций в стиле «новый адаптер вызывает старый» по мере так называемого эволюционного развития системы;
- лоскутная интеграция существующих систем и создание поверх них новых слоёв API.

В результате эволюционного создания Ада Паттернов основной ценностью программиста становится знание, как в данной конкретной системе реализовать даже простую новую функцию, не прибегая к многодневным археологическим раскопкам и минимизируя риски дестабилизации. Код начинает изобиловать плохо читаемыми и небезопасными конструкциями.

Последствия от создания Ада Паттернов ужасны не столько невозможностью быстро разобраться в чужом коде, сколько наличием многих неявных зависимостей.

Но имеются ли у вас в проекте ресурсы для того, чтобы не только обучить всех программистов 150-страничному своду правил, но и постоянно контролировать его исполнение? Поэтому появились новые C-подобные языки: сначала Java, а чуть позже и C#. Они резко снизили порог входа за счёт увеличения безопасности программирования, ранее связанной прежде всего с ручным управлением памятью. Среды времени исполнения Java и .NET решили проблему двоичной совместимости и повторного использования компонентов системы, написанных на разных языках для различных аппаратных платформ.

Эксперты по ООП в своих книгах стали нехотя писать о том, что технология тем эффективнее, чем более идеален моделируемый ею мир. Многоуровневые иерархии классов не воссозданы многолетним трудом классификации объектов окружающего мира, а выращены в виртуальных пробирках лабораторий разработчиков.

Учебники по ООП полны примеров, как легко и красиво решается задачка отображения геометрических фигур на холсте с одним абстрактным предком и виртуальной функцией показа. Но стоит применить такой подход к объектам реального мира, как возникнет необходимость во множественном наследовании от сотни разношёрстных абстрактных заготовок.

Одна из причин подобных злоключений в том, что концепции, выдвигаемые ООП, на самом деле не являются его особенностями за исключением наследования реализации от обобщённых предков с виртуализацией их функций. И по несчастливому стечению обстоятельств именно наследование реализации является одним из основных механизмов порождения ада наследуемых ошибок, неявных зависимостей и хрупкого дизайна. Все остальные концепты от инкапсуляции и абстракции до полиморфизма имеются в вашем распоряжении без ООП. Полиморфизм с проекциями вместо таблиц наличествует даже в SQL.

Особо хочу остановиться на тезисе уменьшения сложности при использовании ООП для создания фреймворков. Современное состояние дел – это платформа. NET с примерно 40 тысячами классов и типов ещё в версии 3.5. Вдумайтесь, вам предлагают для выражения потребностей прикладного программирования язык с 40 тысячами слов, без учёта глаголов и прилагательных, называя такую технологию упрощением.

Надо признать, входной порог использования ООП оказался гораздо выше, чем предполагалось в 1980–90-х гг. С учётом девальвации среднего уровня знаний «прогрессивных технологий», меняющихся по законам бизнеса квазимонополий, и прибывающих выпускников трёхмесячных курсов этот порог ещё и растёт с каждым годом.

С другой стороны, немалое число крупных проектов принципиально не используют ООП, например, ядро операционной системы Linux, Windows API или движок Zend уже упоминавшегося PHP. Язык C по-прежнему занимает первое место согласно статистике активности сообществ программистов44, стабильно опережая второго лидера Java – например, индексы ноября 2012 показывают 19 % против 17 %.

Разумнее предположить, что реальную отдачу от ООП вы получите, только создав достаточно хорошие модели предметных областей. То есть тот самый минимально необходимый словарь и язык вашей системы для выражения её потребностей, доступный не только современным Пушкиным от программирования. Модели будут настолько простыми и ясными, что их реализация не погрузит команду в инфернальные нагромождения шаблонных конструкций и непрерывный рефакторинг, а фреймворки будут легко использоваться прикладными программистами без помощи их авторов. Правда, тогда неизбежно встанет вопрос о необходимости использования ООП…

СУБД, ORM

В технологии отображения объектов на РСУБД есть очень важный момент, от понимания которого во многом зависит успех вашего проекта. Я не раз слышал мнение программистов, что для слоя домена50 генерируемый проектором SQL код является аналогом результата трансляции языка высокого уровня в ассемблер целевого процессора. Это мнение не просто глубоко ошибочно, оно быстрыми темпами ведёт команду к созданию трудносопровождаемых систем с врождёнными и практически неисправимыми проблемами производительности. Проще говоря, как только вы подумали о SQL как о некоем ассемблере по отношению к используемому языку ООП, вы сразу влипаете в очень нехорошую историю.

Обычно разработчикам баз данных я рекомендую избегать необоснованного использования триггеров. Потому что их сложнее программировать и отлаживать. Оставаясь скрытыми в потоке управления, они напрямую влияют на производительность и могут давать неожиданные побочные эффекты. Пользуйтесь декларативной ссылочной целостностью (DRI) и хранимыми процедурами, пока возможно. А если ваш администратор баз данных склонен к параноидальным практикам «запрещено

всё, что не разрешено», избегайте программировать на уровне СУБД, исключая критичные по производительности участки. Приходится говорить это с сожалением…

Мнение

Мне сложно в чём-то согласиться или не согласить с автором. Я ещё ни разу не был задействован в разработке по-настоящему большого проекта с большим количеством разработчиков. Поэтому с описываемыми ужасами я не встречался. Ту разработку, в которой я действительно был задействован, я бы назвал средней по масштабу (или даже полусредней). Но в той разработке код был вполне логично структурирован, то есть архитектура не вызывала замешательств и недоумения. Это не значит, что архитектура была совершенно лёгкой для понимания, но всё же максимально логична (по моему мнению). То есть мне сложно что-то противопоставить мнению автора.

Интересным является мнение автора о возникновении таких языков как Java и C#. Я и раньше понимал, что их целью по сути является упрощение процесса разработки. Но я не понимал, что за этим упрощением стоит желание перенести жизненно-важные вещи с разработчиков на сами технологии. То есть по возможности уменьшить человеческий фактор при разработке. Хотя, опять-таки, откуда создатели самих технологий знают, что они лучше разработчиков справятся с такими вещами? То есть такое стремление, пожалуй, очень хорошо подходит для низко- и среднеквалифицированных разработчик, но не для высококвалифицированных, которые хотели бы сами досконально управлять своим проектом. Таким образом, я не против упрощения процесса разработки, но не в ущерб контролю над своим детищем.

В книге проскакивает очень интересная мысль о том, что технология тем эффективней, чем более идеален моделируемый ею мир (и заметка о том, что ООП этому критерию как раз таки не удовлетворяет). Но не понятно, что подразумевается под идеальным миром. Мир, наиболее приближенный к реальному? То есть, наименее абстрактный мир? Но почему такая технология будет более эффективной? Может потому, что с неабстрактными вещами банально легче работать (ведь мы и так каждый день «работает» с нашим неабстрактным миром). Или потому, что наш мир уже идеально спланирован, и зачем планировать свой, который может быть далеко неидеальным. То есть лучше отталкиваться от чего существующего и реально работающего, чем создавать что-то принципиально новое, рискуя что-то не учесть? Не знаю, на самом деле я не очень понял эту мысль.

По большому счёту автор выражается негативно по отношению к ООП. И главным его доводом в принципе является то, что ООП не оправдало себя как технология, направленная на упрощение. Но мне сложно спорить с автором (или соглашаться), так как я знаком лишь с ООП, то есть сравнивать мне не с чем. Хотя и раньше я часто слышал мнение, что часто использование ООП неоправданно, так как часто без него может выйти проще (то есть мы можем столкнуться с неоправданным усложнением, а вовсе не с упрощением). Короче говоря, будущее не за ООП (наверное).

Про СУБД и вовсе нечего сказать. Автора понесло в сложные для понятия и неизвестные технологии. Сам я знаком с базами данных достаточно поверхностно, хотя сам участвовал в разработке конкретной базы и программного слоя для работы с ней. И мы использовали фреймворк Hibernate, столкнувшись при этом с довольно серьёзными проблемами. Да и вообще, эта часть оказалась для меня самой сложной и проблематичной во всём проекте. И по сути, вся проблема состояла в том, что было практически невозможно соотнести модели самой базы с моделями в самой программе. Мы столкнулись с тем, что нам приходилось идти на уступки (фактически на ухудшение качества) либо в самой БД (нарушать нормы), либо в коде (делать велосипедные костыли). И мы выбрали второе, ибо на момент осознания проблемы БД уже была целиком сформированной и довольно объёмной. В том смысле, что она содержала относительно много сущностей, связи между которыми изначально были запланированы для обеспечения всего функционала приложения (и это всё при выполнении трёх норм). Поэтому как-либо портить БД очень хотелось. А портить код всегда пожалуйста (шутка).

Таким образом, хоть мой опыт и не идёт ни в какое сравнение с опытом автора, я, пожалуй, могу понять то, о чём он говорит.

Часть 5. ВЦКП и прогресс

Можете ли вы представить себе личный автомобиль, передвигающийся во время поездок с предусмотренной скоростью лишь 1 час из 10? Именно таков ваш персональный компьютер, планшет или смартфон. Его вычислительная мощность используется в среднем на 5–10 % даже на рабочем месте в офисе. Более пропорционально расходуется оперативная память. Операционная система Windows NT 4 свободно работала на устройстве с ОЗУ66 объёмом 16 Мбайт. Windows 7 требуется уже минимум 1 Гбайт. Правда, я не уверен, что Windows 7 хотя бы по одному параметру в 60 раз лучше предшественницы.

Широко известный в узких кругах своими несколько провокационными книгами67 публицист Николас Карр пишет: «Отделы ИТ в компаниях уходят в прошлое, а сотрудники соответствующих специальностей останутся не у дел из-за перехода их работодателей на сервис, предоставляемый по принципу коммунальных услуг»

Чем занимались тогда проектировщики, техники и организаторы? Примерно тем же самым, чем занимаются сейчас продвигающие на рынок системы «облачных» (cloud) вычислений, за исключением затрат на рекламу. Создавали промышленные вычислительные системы, отраслевые стандарты и их реализации. Чтобы предприятия-пользователи, те, кому это экономически нецелесообразно, не содержали свои ВЦ, а пользовались коллективными, ведь пироги должен печь пирожник, а не сапожник.

В чем состояли просчёты советской программы ВЦКП? Их два. Один крупный: проглядели стремительную миниатюризацию и, как следствие, появившееся обилие терминалов. Хотя отдельные центры, как, например, петербургский «ВЦКП Жилищного Хозяйства», основанный в 1980 году, действуют до сих пор, мигрировав в 1990-х от мейнфреймов к сетям ПК. Второй: просчитались по мелочам, не спрогнозировав развал страны с последующим переходом к состоянию технологической зависимости.

Наиболее очевидное применение децентрализации – автономные программы аналитической обработки данных. Мощность терминала позволяет хранить и обрабатывать локальную копию части общей базы данных, используя собственные ресурсы и не заставляя центральный сервер накаляться от множества параллельных тяжёлых запросов к СУБД.

Значит ли это, что софтостроительные фирмы теперь, как утверждают некоторые маркетологи, «производят услуги»? Разумеется, нет. Во-первых, услуги не производят, их оказывают. Во-вторых, услуга от продукта, материального товара, принципиально отличается минимум по трём пунктам:

- услуги физически неосязаемы;
- услуги не поддаются хранению;
- оказание и потребление услуг, как правило, совпадают по времени и месту.

В схеме «программа как услуга» посредник (ВЦКП) берет на себя эксплуатацию программного продукта, то есть его установку, настройку, содержание, обновление и т. п., предлагая конечному пользователю услугу по доступу к собственно функциональности этой программы.

Большие ЭВМ до сих пор очень важны (для облачных вычислений опять-таки).

Произошла тихая революция. Ещё 15–20 лет назад сессии крупных поставщиков на конференциях разработчиков были своеобразным мастер-классом, где на бета-стадии испытывалась реакция аудитории на предлагаемые изменения. Сегодня повестка дня состоит в постановке перед фактом новой версии платформы, показе новых «фишек» и оглашении списка технологий, которые больше не

будут развиваться, а то и поддерживаться. Действительно, солдаты от софтостроения не должны рассуждать. Они несут службу и должны молча овладевать оружием, закупкой которого занимаются генералы в непрозрачном договоре с поставщиками. Экономика потребления обязана крутиться, даже если в ней перемалываются миллиардные бюджеты бесполезных трат на модернизацию, переделку и переобучение.

Мнение

Интересно было почитать про ВЦКП (и про их современный аналог – облачные вычисления). Хотя, полагаю, всё это относится к промышленному и корпоративному использованию. Вряд ли рядовому пользователю нужны такие средства. И совсем непонятно, к чему был рассказ о возросшей и неиспользуемой мощности личных терминалов. Как это относится к облачным вычисления, если речь идёт как раз-таки о перекладывании работы с личных терминалов на внешние сервера? Такое чувство, что автор просто не определился, о чём говорить в главе.

Интересен рассказ автора о «новых технологиях». Меня лично удивило заявление, что новые технология штампуются (даже когда в этого один вред) лишь для заманивания потребителя, то есть в маркетинговых соображениях. Хотя, такое действительно можно заметить. Например, продукция компании Apple, которая уже давно высасывает «новые технология» откуда попало и делает из этого революцию. И ведь народ ведётся.

Часть 6. Проектирование

«Качество появляется только тогда, когда кто-нибудь несёт ответственность лично».

— Фредерик Ф. Брукс

Тонким клиентом (thin client) традиционно называют приложение, реализующее исключительно логику отображения информации. В классическом варианте это алфавитно-цифровой терминал, в более современном – веб-браузер.

В противоположность тонкому, толстый клиент (rich client) реализует прикладную логику обработки данных независимо от сервера. Это автономное приложение, использующее сервер в качестве хранилища данных. В трёхзвенной архитектуре толстым клиентом по отношению к СУБД является сервер приложений.

Так называемый «умный» (smart client) клиент по сути остаётся промежуточным решением между тонким и толстым собратьями. Будучи потенциально готовым к работе в режиме отсоединения от сервера, он кэширует данные, берет на себя необходимую часть обработки и максимально использует возможности операционной среды для отображения информации.

Не секрет, что возможности отображения у веб-браузера, как программируемого терминала, очень скромные, по сравнению с автономным приложением. Компромиссным решением является так называемое «насыщенное интернет-приложение», также являющееся тонким клиентом, но обладающее всеми возможностями отображения клиента толстого.

Даже в простой программе типа записной книжки имеется минимум 2 уровня:

- Уровень приложения, реализующий функционал предметной области.
- •Уровень служб, поддерживающих общую для всех разрабатываемых приложений функциональность. Например, метаданные, безопасность, конфигурация, доступ к данным и т. д.

В свою очередь, общие для многочисленных служб функции группируются в модули и соответствующие API уровня ядра системы.

Кто не хочет – ищет причины, кто хочет – средства. Ищите возможности сократить путь информации от источника к пользователю и обратно.

Если вы дочитали предысторию до конца, вдумайтесь в цифры. КИС, реализующая основные функции автоматизации деятельности торгово-производственной фирмы среднего размера: от бухгалтерии и складов до сборочного производства и сбыта – была разработана командой из 4–5 человек примерно за полтора года, включая миграцию с предыдущей версии. Система критичная, даже короткий простой оборачивается параличом деятельности фирмы. Причина столь сжатых сроков? Ясное понимание решаемых прикладных задач, создание соответствующего задаче инструментария, прежде всего, языка бизнес-правил высокого уровня, и подтверждение тезиса Брукса о многократно превосходящей производительности хороших программистов по сравнению с остальными.

Последние годы в ходе аудита баз данных я не раз наблюдал, как современные команды в 2–3 раза большей численности, вооружённые умопомрачительными средствами рефакторинга и организованными процедурами гибкой разработки, за год не могли родить работоспособный заказной проект, решающий несколько специфичных для предприятия задач. Сотни тысяч строк кода уходили в мусорную корзину или продолжали поддерживаться с большими трудозатратами, сравнимыми с переделкой.

Мнение

С рассуждениями о проектировании сложно не согласиться. Всё же человек с опытом и явно знает, что говорит. Единственное, хотелось, чтобы он подробней описал второй (из двух упомянутых) уровней любого приложения, а именно уровень служб, поддерживающих общую для всех разрабатываемых приложений функциональность. Собственно, интересно узнать мнение автора о том, как этот уровень строится. Ибо понятно, что речь идёт о каких-то готовых, распространённых решениях. Но до какой степени хорошо использование готовых решений? С одной стороны, конечно же никто не будет каждый раз все повторяющиеся аспекты делать с нуля. У самого разработчика и без популярных сторонних решений со временем сформируются свои. Но и в рамки себя загонять не хочется, ибо они могут пагубно повлиять на сам продукт (хотя в творческом плане). И сложно разглядеть эту грань. Грань между оправданным упрощением своей работы и соглашением на рамки (внешние, либо свои собственные).

То есть, насколько вообще проект должен быть личным? Например, в том же кинематографе никто не любит плагиата. Это сразу бросается в глаза, и это сразу осуждается. Но если позаимствовать элементы из других произведений и при этом уместно и правильно их применить, при этом добавив что-то своё (при чём не обязательно свои элементы, а может всего лишь свой подход, своё виденье, свою любовь и уважение к ремеслу), то вполне может получиться шедевр. (Вспомнить того же Квентина Тарантино).

Я к тому, что разработка – это творчество. И об этом не надо забывать. Но также это и технический труд. То есть, никто не потребует от вашего фильма отсутствия багов и поддержку Internet Explorer 5. А от проекта потребуют. В этом и сложность, как найти это тонкое равновесие?

И на конкретном примере (пример создания КИС для компании «Ниеншанц») автор показывает, что иногда лучше иметь способность создавать свои собственные инструменты, а не обладать навороченными сторонними. Так же автор утверждает, что лучше быть хорошим программистом, чем не очень хорошим (правильная, но не особо изящная мысль).

Я считаю профессию разработчика очень сложной. Делать выбор в жизни всегда сложно. А когда твоя работа состоит из постоянно выбора «А как сделать лучше всего?», то это совсем тяжко. Но такие люди нужны. Всегда нужны те, кто примет на себя ответственность, кто решит, как сделать лучше, и укажет другим. А всё потому, что людям надо, чтобы им управляли. Надо, чтобы за них делали выбор, ибо опять-таки, выбирать – это тяжело. И так во всех сверах, вплоть до политики и власти. И тут мы приходим к неоднозначному выводу. Если ты хочешь быть кому нужен, то будь в чём-то хорош. И

вроде как это должно мотивировать (собственно и сам автор, по сути, рекомендует задуматься над своей компетентностью), но я не знаю, хорошо ли это, или плохо.

Опять-таки, конечно же человек должен добросовестно выполнять ту работу, которой занимается. Просто человеческое общество так построено. Человек всегда выживал благодаря своим навыкам (и в первобытные времена, и в современны). А сейчас тем более, ведь теперь никто не хочет просто выживать. Все хотят быть признанными, значимыми, необходимыми. Но является ли человек просто набором умений и характеристик? Если да, то всё отлично, ведь само общество требует от тебя их наращивание. А если нет? Не теряем ли мы самих себя в устройстве нашего общества?

Лично я не хочу быть просто квалифицированным специалистом, просто набором навыков. Просто человеком, который качественно сделает для вас приложение. Я считаю, что идеальное общество, это общество, в котором людям не придётся выживать. Только так человек может пойти дальше в своём развитии. Сейчас же это доступно, пожалуй, только очень богатым людям. А может я ошибаюсь. Может для кого-то повышение своей квалификации укладывается в рами саморазвития (что безусловно важно). И конечно же, даже если людям не пришлось бы зарабатывать себе на жизнь, многие всё равно бы занимались бы той же разработкой. Всё так же создавали бы для других людей приложения, приобретали бы навыки и так далее. В общем, тут есть о чём подумать.

Часть 7. Шаблонное мышление

Техническая культура – это не производства и знания, а люди, умеющие это делать и применять.

Прикладная разработка не место для эстетического наслаждения от красот технологий. Это бизнес.

Архитектурные эксперименты оправданы, если маркетолог видит преимущества в архитектуре товара, которые преобразуются в удовлетворение потребностей клиентов. Надо оценивать эффективность архитектуры не в терминах красот, а в терминах трудозатрат. Пользователю все равно, он не видит, что там внутри. Но он видит, что система дорабатывается медленно или валится с ошибкой.

Прочтение сего труда новичком, как мне кажется, является прямым аналогом попадания в прокрустово ложе диктуемой парадигмы. Потому что книга описывает набор решений, а неокрепшему за недостатком практики уму проектировщика надо научиться самому находить такие решения и пути к ним. Для чего гораздо эффективнее первое время «изобретать велосипеды», нежели сразу смотреть на готовые чужие. На чужие надо смотреть, когда придуман хотя бы один собственный, чтобы понять, насколько он несовершенен, и выяснить, каким же путём можно было бы прийти к лучшим образцам велосипедов данной модели.

Однако книга все-таки дала всплеск, и по воде пошли круги. А. Алексан-дреску в книге «Modern C++ design» начал экспериментировать с реализацией шаблонов на C++ и продвигать в массы свои велосипеды. Появились издания с достаточно абсурдными названиями. Например, «Шаблоны на C#». Постойте, коллеги, какие ещё шаблоны на C#? Оказывается, шаблоны сыграли с программистским сообществом злую шутку: они оказались ещё и зависимыми от языка программирования. То есть, прочитав «банду четырёх», писавших свой труд исходя из возможностей C++, срочно бегите за новой порцией информации, кто для Java, кто для C#. Хотя задумывались шаблоны с обобщающей практики целью.

Итого на простом примере имеем целый букет решений вместо одного шаблонного, из которых можно выбирать оптимальный. Попытавшись однажды объяснить подобное программисту, видимо, несколько увлёкшемуся шаблонами, я не встретил понимания.

Несколько позднее я наткнулся в магазине на книгу с названием, претендующим на звание наиболее абсурдного из встречавшихся. Оно звучало как «Thinking in patterns». В переводе на русский язык – «Мыслить шаблонно». Ещё недавно привычка шаблонно мыслить считалась в инженерном сообществе признанием ограниченности специалиста, наиболее пригодного для решения типовых

задач с 9 утра до 6 вечера. Теперь не стесняются писать целые книжки о том, как научиться шаблонному мышлению…

ОСНОВНОЕ ПРАВИЛО

Обобщаемые классы должны иметь сходную основную функциональность.

Например, если два или более класса:

- порождают объекты, реализующие близкие интерфейсы;
- занимаются различающейся обработкой одних и тех же входных данных;
- предоставляют единый интерфейс доступа к другим данным, операциям или к сервису.

Почему шаблоны «вдруг» стали ненужными? Потому что требуется только обобщение, причём в перечисленных случаях необходимость операции более чем очевидна. Требования же исходят напрямую из вашей задачи. Корректно проведя обобщение, вы автоматически получите код и структуру, близкую к той, что вам предлагают зазубрить и воспроизводить авторы разнообразных учебников шаблонов.

ВОЗЬМИТЕ ЗА ЭМПИРИЧЕСКОЕ ПРАВИЛО

Глубина более двух уровней при моделировании объектов предметной области, вероятнее всего, свидетельствует об ошибках проектирования.

Мнение

На самом деле, согласен со всем, что сказано автором про шаблонное проектирование. Я думаю, что всем должно быть очевидно, что думать надо своей головой. Не самый лучший вариант начинать с шаблонов. Программист должен идти своим путём, вырабатывая свой стиль, своё виденье, лишь опираясь на что-то (на те же шаблоны, например). То есть, тут автор совершенно прав. К тому же, проскакивает интересное рассуждение о том, что шаблоны могут сформировать неправильно представление о сложности каких-либо элементов (задач, методов и т.д.). То есть, попытка что-то формализовать и структурировать может это что-то ненамеренно усложнить (усложнить для восприятия и осознания). То есть лучше проделать свой, (возможно, долгий и неэффективный) путь, чтобы выстроить себя как специалиста. Собственно, по жизни так во всём.

Часть 8. Зри в корень

Дам совет начинающим: учите сиквел, транзакции, уровни изоляции, и будет ваша система быстрой и надёжной. И не только в примере с учётной системой, но и вообще по жизни. Опытным же разработчикам есть поле для оптимизации и дальнейшего совершенствования решений, включая альтернативные подходы.

Что же заставило учёных отказаться от системы Птолемея? Ответ с позиций современной теории познания мы видим в следующем. Система Птолемея описывала движение планет, видимое с Земли, то есть описывала явление. Если же мы оказались, например, на Меркурии или Марсе, то земную птолемеевскую систему нам пришлось бы упразднить и заменить новой.

Система Коперника сумела схватить сущность взаимного движения планет Солнечной системы. Такое описание, говоря современным языком, уже не зависело от того, какую планету в качестве системы отсчёта захочет выбрать себе наблюдатель. С точки зрения теории познания объективной истины теологи совершали грубейшую ошибку: они сущность подменяли явлением. Наблюдаемое с Земли движение планет по небосводу они считали их действительным движением относительно Земли.

Явление зависит от условий наблюдения. Сущность от этих условий не зависит.

Мнение

Опять-таки сложно не согласиться с автором. Но собственное весь прочитанный отрывок можно ужать до определённой мысли: зри в корень. И это похвально. Мне кажется, что автор пытается заставить нас быть разумней, так сказать. Я и сам с таким сталкивался. Когда кажется, что что-то понял (какую-нибудь вещь или систему), но потом оказывается, что недостаточно глубоко и точно. То есть, очень важным качеством я считаю умение видеть суть, вкапываться вглубь, чтобы действительно что-то осознать. Мне кажется, что люди часто просто не осознают степень их непонимания тех вещей, которые, как они думают, они понимают. И так во всех сферах жизни. Не только при проектировании чего-либо. Так что, ещё один плюсик в копилку этой книги.

Часть 9. Компетентность

Что имеем по итогам более чем 10-летнего развития технологий? Появилась возможность стандартизировать хранение и доступ к большому массиву данных, используя вполне обычное серверное оборудование корпоративного класса и 64-разрядную промышленную СУБД. Возникла необходимость переделывать программное обеспечение из-за утраты поддержки среды разработки поставщиком и соответствующих компетенций на рынке труда. Но я совсем не уверен, что ещё через 10 лет новые подрядчики, вынужденные в очередной раз переписывать систему, найдут документацию в том же полном виде, в котором нашли её мы. Если вообще найдут хоть что-то, кроме кода, остатков презентаций и наших отчётов.

Архитектура подобных систем, можно сказать, типовая. Проектируем специализированное хранилище, ищем в корпоративной среде источники нужной информации, организуем регулярное обновление данных, предоставляем пользователям интерфейс для доступа к данным: непосредственно к хранилищу или к так называемым витринам (datamart) и кубам. Наибольшую трудность, кстати, вызывает не сам импорт данных, а поиск в компании людей, которые могут знать, где эти данные взять. Всё-таки 20 тысяч таблиц SAP R3 и примерно столько же в корпоративном хранилище – не шутка, поэтому без хороших проводников раскопки источников информации обернутся поиском иголки в стоге сена.

А выступающий на сцене ведущий эксперт не постеснялся напрямую высказать призыв: «Последние годы я вижу тотальное падение компетенции в области баз данных. DBA, проснитесь!»

Хуже, когда вполне программистский коллектив умудряется годами работать без системы контроля версий исходников, и тогда в коде половину объёма составляют закомментированные куски многолетней давности. Выбросить их жалко, вдруг пригодятся. Но и контроль версий с архивацией не спасает от цифровой пыли десятилетий. В подобных залежах порой можно обнаружить настоящие образцы софтостроительных антипрактик.

Мне хотелось лишь донести простую мысль, что ревизия кода, несомненно, весьма полезная процедура, но как минимум при двух условиях:

- эта процедура регулярная и запускается с момента написания самых первых тысяч строк;
- процедуру проводят специалисты, имеющие представление о системе в целом. Потому что отловить бесполезную цепочку условных переходов может и компилятор, а вот как отсутствие контекста транзакции в обработке повлияет на результат, определит только опытный программист.

Дж. Фокс выводит из своего опыта проектной работы в IBM важную мысль, что большой ошибкой является привлечение к процессу внутреннего тестирования и обеспечения качества посредственных программистов. По его мнению, компетентность специалиста в этом процессе должна быть не ниже архитектора соответствующей подсистемы. Действительно, ведь оба работают примерно на одном уровне, просто один занят анализом, а другой – синтезом.

Качество кода во многом зависит от степени повторного использования, поэтому приведу простой и доступный способ проверки того, не занимается ли ваша команда программистов копированием готовых кусков вместо их факторизации. Для этого регулярно делайте сжатый архив исходников, например, zip с обычным коэффициентом компрессии, и оценивайте динамику роста его размера относительно количества строк. Если размер архива растёт медленнее, чем количество строк, это означает рост размера кода за счёт его копирования.

Мнение

Автор рассказывает о том, что стремительное развитие технологий может нести серьёзные сложности, ибо частые переходы серьёзных программных комплексов на новые технологии могут быть очень болезненными. Опять-таки мы возвращаемся к проблеме дальновидности. Люди не хотят смотреть вперёд, не ходят думать о последствии своих действий, насколько серьёзными они бы не были. И так во всех сверах нашего общества.

Но, когда речь касается развития IT-технологий, то, наверное, иначе быть и не может. Разработчики конечно же могут поступать умно и дальновидно, но развитие технологий ведь не остановишь. Я думаю, что основным стимулом развития программных технологий является развитие аппаратных технологий (само собой, это не единственный стимул). То есть, чем совершеннее машина, тем более развитый софт для неё можно писать.

И опять-таки поднимается вопрос о компетенции разработчиков. Собственно, вся книга по сути об этом.

Часть 10. Кризис и культура

Не все гигагерцы и гигабайты расходуются впустую. Кризис в софтостроении, о котором говорят уже более 30 лет, продолжается. В ответ на усложняющиеся требования к программным системам и неадекватные им методологии (технологии), особенно в части моделирования и проектирования, индустрия выставила свое решение. Оно состоит в достижении максимальной гибкости средств программирования и минимизации ошибок кодирования. Проще говоря, если мы не можем или не успеваем (что в итоге приводит к одному и тому же результату) достаточно хорошо спроектировать систему, значит, надо дать возможность быстро и с минимальными затратами её изменять на этапе кодирования. Но принцип для заказчика остался прежним: «Быстро, качественно, дёшево – выбери два критерия из трёх».

Очень важно отделить редкую ситуацию «бизнес меняется еженедельно» от гораздо более распространённой «представления команды разработчиков о бизнесе меняются еженедельно». Если вам говорят о якобы часто изменяющихся требованиях, всегда уточняйте, о чём, собственно, идёт речь.

Необходимо отличать спиральную модель от итеративной. Спиральная модель сходится в точку «система готова», итеративная модель в общем случае не сходится, но обеспечивает реализацию всё новых и новых требований.

Ключевой особенностью спиральной технологии является прототипирование. В конце каждого витка после этапа стабилизации заказчик получает в своё распоряжение ограниченно работающий прототип целой системы, а не отдельных функций. Основная цель прототипа состоит в максимально возможном сближении взглядов заказчика и подрядчика на систему в целом и выявлении противоречивых требований.

Взаимное перекладывание ответственности на сложных проектах.

Качество программного продукта – многозначное и сложное понятие. Производственная культура – ещё более сложное. В одном можно быть уверенным: ни о какой культуре софтостроения не может идти и речи, если любой программист из коллектива не способен остановить бессмысленный

циклический процесс для выяснения, какого же рожна по историям заказчика потребовалось обобщать четырёхногих коров и обеденные столы.

Мнение

Наконец-то мы узнали различия между спиральной итерационной моделями разработки. А в остальном мне нечего сказать. Ибо весь опыт разработки и тестирования, который у меня есть, вполне является поверхностным. Поэтому спорить с автором (а иногда и понимать) я не могу.

Часть 11. Генерация кода

Идея разрабатывать программы, минимизируя стадию кодирования на конкретных языках под заданные платформы, появилась достаточно давно. Прежде всего в связи с неудовлетворительной возможностью языков высокого уровня третьего поколения (3GL) описывать решаемые прикладные задачи в соответствующих терминах. За последнее время к этой причине добавилась ещё и поддержка независимости от целых платформ, ведь прогресс, как мы знаем, неотвратим, особенно «прогресс».

В управляемой моделями разработке (УМР) и в программной фабрике в частности наиболее интересной возможностью является генерация кода, скомпилировав который, можно сразу получить работающее приложение или его компоненты. Мы проектируем и сразу получаем нечто работающее, пусть даже на уровне прототипа. Уточняя модели, мы на каждом шаге имеем возможность видеть изменения в системе. Проектирование становится живым процессом без отрыва от разработки.

Это означает, что оснащённый средствами автоматизации программист с навыками моделирования на этапе разработки рутинного и специфичного для платформ/архитектур кода производителен примерно так же, как и его 50 коллег, не владеющих технологией генерации кода по моделям. Любое внесение изменений в модель тут же приводит в соответствие все генерируемые слои системы, что ещё более увеличивает разрыв по сравнению с ручными модификациями. Наконец, для генерируемого кода не нужны тесты. Производительность возрастает ещё как минимум вдвое. Даже если принять во внимание, что доля рутинного и прочего инфраструктурного кода по отношению к прикладному, то есть решающему собственно задачи конечных пользователей, снижается с масштабом системы, есть о чём поразмыслить в спокойной обстановке.

Мнение

И снова автор рассказывает о том, как лень и недальновидность разработчиков (а также неповоротливость крупных компаний) приводит к неоправданно большому количеству лишней работы. И снова мы обещаем себе быть хорошими разработчиками.

Поднимается вопрос автоматической генерации кода (например, из графических диаграмм). И это, конечно, интересный вопрос. Автор совершенно правильно (ещё бы) выделяет проблемы и недостатки современных систем такого рода. Но что будет, если эти недостатки исчезнут. Если эти системы станут совершенными и будут генерировать идеальный код (хотя не факт, что такое вообще возможно). Ясно, что в таком случае кодеры исчезнут, как таковые. Но насколько изменится профессия разработчика? Полагаю, что она станет более доступной, снизится порог вхождения.

И ведь такие тенденция, по всей видимости, наблюдаются с самого появления профессии разработчика и IT-сферы вообще. С развитием технологии становятся всё проще для работы и применения. Так и сами программисты из университетской элиты постепенно стали вполне рядовыми рабочими. И что, если в будущем их самих заменят программы, которые будут писать программы? Так и в сфере промышленности люди прошли путь от ремесленников до промышленных роботов.

С древних времён люди применяют известные им знания и технологии для упрощения своего труда, вплоть до того момента, пока эти самые технологии полностью не заменяют человека. Ждёт ли то же самое разработчиков и программистов? Верно ли говорить о деградации профессии? Или всё-таки об её развитии? Мне, на самом деле, хочется быть оптимистом в этом вопросе. Сам я считаюсь будущим

разработчиком, и не хочется думать, что я, не начав свой путь, уже стану ненужным. К тому же я уверен, что многие скажут мне, что человека не так уж и легко заменить в настолько интеллектуальных занятиях, как разработка чего-либо (но ведь ИИ не дремлет).

Но в общем, я считаю, что это правильный путь. По моему мнению, человечество должно достичь такой стадии, когда человек сможет всю свою жизнь заниматься, скажем так, возвышенными и полезными вещами. А именно: творчеством, исследованиями, преподаванием, развитием и т.д. и т.п. То есть человек не должен стоять за станком, производя продукты потребления. Человек, будучи существо, способным осознавать себя и окружающий мир, должен заниматься чем-то более важным.

Но, возможно разработка как раз-таки является чем-то важным. Всё-таки она вполне связана с исследованиями. Да и является творческим процессом, как мы выясняли раньше. В общем, я не берусь однозначно отвечать на этот вопрос.

При этом, в тексте есть оговорки про то, что средства автоматизации призваны не к тому, чтобы лишить разработчика работы, а к тому, чтобы избавить его от рутины, оставив больше времени для действительно профессионального и творческого труда. Как говорится, рутина убивает. Если смотреть на подобные системы в таком свете, то, пожалуй, все разработчики должны обеими руками быть за них.

Часть 12. Дефрагментация мозга

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». Это ещё далеко не полный хаос, но уже давно и далеко не порядок.

Одна из проблем организации промышленного производства программного обеспечения состоит в отсутствии каких-либо формальных описаний деятельности программиста. Можно определить в технологической карте, как работает сварщик или каменщик, но как пишет программу программист, зачастую, не знает и он сам. До художника, конечно, далеко не все дотягивают, а вот с деятельностью рядового журналиста «независимой» газеты непосвящённому в софтостроение человеку сравнивать вполне можно. Этакий ядрёный сплав ремесла, некого богемного искусства, со вкраплениями науки, вперемешку с халтурой, шабашкой и постоянным авралом. Попытки же принудить программиста делать однотипные операции противоречат самой цели существования программного обеспечения как самого гибкого из существующих средств автоматизации рутинных процессов и потому изначально обречены на неудачу.

Кстати, если программист говорит вам, что в данном месте программа «должна работать», это значит, что с очень большой вероятностью она не работает, а он её просто не проверял в этом месте. Если же программист уже после обнаружения ошибки говорит: «А у меня она в этом месте работает…», лучше сразу его уволить, чтобы не мучился. Это проза жизни, также относящаяся к коллекции моих практик. (Как-то жестоко).

Почему же относительно легко студентам? Дело, прежде всего, в смене образа мыслей, а может быть, и восприятия самой действительности. Мир – это такая очень большая и сложная компьютерная игра, а преподаватели, соответственно, должны обучать не премудростям стратегий познания мира, а практическим приёмам, секретным кодам и даже шулерству, чтобы пройти в этой игре на следующий уровень. Этакие «мастера ключей» из Матрицы. Про то, что в игре продукты на столе появляются прямо из холодильника, тоже стоит упомянуть.

Является ли фрагментарное мышление приспосабливанием к многократно увеличившемуся потоку информации? Отчасти да, только это скорее не адаптация, а инстинктивная защита. Чтобы осознанно фильтровать информацию о некоторой системе с минимальными рисками пропустить важные сведения, нужно иметь чётко сформированные представления о ней, её структуре и принципах

функционирования. Если, например, у стажера нет знаний о СУБД в целом, то курс по SQL Server – конкретной её реализации, сводится к запоминанию типовых ситуаций и решений из набора сопутствующих практических работ. Вся теоретическая часть при этом просто не проходит фильтры.

В такой ситуации альтернативой ухода от информационных потоков и некачественного образования во фрагментарную реальность является самообразование на базе полезных книг. Потому что хорошая книга – самый эффективный способ дефрагментации ваших мозгов. Если не верите, возможно, вас убедит рассказ классика научной фантастики А. Азимова «Профессия» о неудавшемся программисте.

Итак, настоящий исследователь должен:

- быть достаточно ленивым. Чтобы не делать лишнего, не ковыряться в мелочах;
- поменьше читать. Те, кто много читает, отвыкают самостоятельно мыслить;
- быть непоследовательным, чтобы, не упуская цели, интересоваться и замечать побочные эффекты.

Последнюю главу книги можно целиком внести в конспект.

Мнение

Поднимается вопрос о сложности описания работы программиста. То есть, о сложности её систематизации. И, наверное, это связанно с тем, что профессия программиста не является, так сказать, односложной. То есть, часто в профессиях можно описать чёткий алгоритм действий. А в программисты, хоть уже и являются рядовыми сотрудниками, но всё же выполняют более комплексную работу. То есть, даже самый рядовой программист постоянно принимает решения, непосредственно связанные с тем, чем он занимается. То есть, можно сказать, что сам процесс программирования – это постоянное проектирование (но, конечно же, не такого уровня, как проектирование всего проекта). В этом, к слову, и большой плюс профессии программиста: профессия одна, но каждый раз всё новое.

Очень занятным является сравнение софтостроения и писательства. Собственно, с тем, что они схожи, я согласен. Ведь что такое программирование? По сути, это составление чего-то большого и сложного из множества чего-то маленького и простого. То есть, взять какие-нибудь простейшие объекты, которые выполняют простейшие действия, понять, как они могут взаимодействовать, и выстроить из них систему, способную делать что-то сложное. В этом и проявляется красота, или искусность. И ведь в литературе то же самое. Только, если в программировании мы берём функции, классы, библиотеки и т.д., то в литературе мы берём эпитеты, речевые обороты, сюжетные повороты и т.д. Если в программировании мы выясняем, что делают функции, какие методы есть у классов, как всё это применить вместе, то в литературе мы думаем, как выстроить характеры персонажей, как сохранить логику событий, как читатель отнесётся к тому или иному сюжетному повороту. И при всём этом, цель у обоих процессов одна: составить общую, целостную, рабочую картину. То есть, пожалуй, можно сказать, что книги вполне себе проектируются, а программы вполне себе пишутся (в той или иной степени).

К слову, вот книга и прочитана. И на самом деле, после прочтения осталось желание вернуться к этой книге позже (возможно намного позже), потому что есть ощущение, что я понял далеко не всё (наверняка понял далеко не всё), что хотел сказать автор. Полагаю, нам ещё предстоит узнать на собственном опыте о том, в каком состоянии сейчас находится IT-сфера. А также точнее понять, какой путь она прошла. То есть надеюсь, что когда-нибудь нам удастся составить в голове целостную картину всего этого. Что касается меня, пока что это точно не так. Но фундамент однозначно положен.

Грушевский Андрей

Программист – это человек, способный заставить компьютер решать поставленное перед ним множество задач.

Пока одни корпели над программами и моделями в кружках, другие реализовывали свои интересы, вполне возможно, творческие, но не технические. И вот в связи с процессом заполнения сферы услуг, о котором мы ещё поговорим, эти другие тоже оказались в софтостроении, поскольку имеется устойчивый спрос на рабочую силу. Разумеется, для таких работников главной, а то и единственной целью будет сделать «чтобы как-то работало». Это даже не ремесло в чистом виде, а неизбежные плоды попыток индустриализации в виде халтуры и брака.

Софтостроение на текущий момент – эклектичный сплав технологий, которые могут быть использованы как профессионалами технического творчества, так и профессионалами массового производства по шаблонам и прецедентам.

Весьма показательный уровень программирования в одной социальной сети можно было оценить по пришедшему от их имени письму следующего содержания: «Ваши фотографии были перенесены на наш новый фотохостинг. Всего было перенесено 0 фотографий». Для того чтобы вставить в код программы рассылки проверку IF > 0, нужно, видимо, иметь не только недюжинные умственные способности, но и дополнительную квалификацию, равно как и понимание сути выполняемой задачи. С другой стороны, да и чёрт с ними, с сотнями тысяч отправленных бесполезных писем. Одной массовой рассылкой больше, одной меньше, не правда ли?

Для начинающих я составил небольшой словарик ключевых фраз, часто присутствующих в объявлении о вакансии. По замыслу, он должен помочь молодому соискателю вакансии программиста разобраться в ситуации и принять решение на основе более полной информации:

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.

Я сформулировал бы основные принципы хорошего резюме следующим образом:

- Краткость – сестра таланта. Даже в небольшой фирме ваше резюме будут просматривать несколько человек. Вполне возможно, что первым фильтром будет ассистент по кадрам, который не имеет технического образования и вообще с трудом окончил среднюю школу. Поэтому постарайтесь на первой странице поместить всю основную информацию: ФИО, координаты, возраст, семейное положение, мобильность, личный сайт или блог, описания своего профиля, цель соискания, основные технологии с оценкой степени владения (от «применял» до «эксперт»), образование, в том числе дополнительное, владение иностранными языками. Всё остальное поместите на 2–3 страницах.
- Кто ясно мыслит, тот ясно излагает. Все формулировки должны быть ёмкими и краткими. Не пишите «узнавал у заказчика особенности некоторых бизнес-процессов в компании» или «разработал утилиту конвертации базы данных из старого в новый формат». Пишите «занимался постановкой задачи» или «обеспечил перенос данных в новую систему».
- Не фантазируйте. Проверьте резюме на смысловые нестыковки. Если на первом листе значится «эксперт по C++», но при этом в опыте работы за последние 5 лет эта аббревиатура встречается один раз в трёх описаниях проектов, то необходимо скорректировать информацию.
- Тем более не врите. Вряд ли кадровики будут звонить вашим предыдущим работодателям, но софтостроительный мир тесен, а чем выше квалификация и оплата труда, тем он теснее. Одного прокола будет достаточно для попадания в «чёрный список» компании, а затем через общение кадровиков и агентств по найму – ещё дальше.
- Если соискание касается технического профиля, в каждом описании опыта работы упирайте на технологии, если управленческого – на периметр ответственности, если аналитического – на разнообразие опыта и широту кругозора.
- Не делайте ошибок. Пользуйтесь хотя бы автоматической проверкой грамматики. Мало того, что ошибки производят негативное впечатление, они могут радикально изменить смысл фразы. Например, если написать «политехнический университет»
- Будьте готовы, что далее первой страницы ваше резюме читать не станут, а о подробностях «творческого пути» попросят рассказать на первом собеседовании.

Один из этапов поиска работы – отбрыкаться от приглашения на бесполезное интервью, убедив, что не стоит зря тратить время. Ведь у мадемуазелек рабочего времени навалом. Достаточно попросить прислать краткое описание требований к вакансии. В 90 % случаев это срабатывает, причём в 90 % из этих 90 % случаев требования оказываются неподходящими. В оставшихся 10 % можно соглашаться на интервью, главное – потом накануне не забыть уведомить, что по уважительной причине прийти на него нет никакой возможности.

Термин «гуглизация» (googlization) не случайно созвучен с другим, с глобализацией. И если глобализация систематично уничтожает закрытые экономики, то «гуглизация» нивелирует энциклопедические знания. Пространство практического применения эрудита сузилось до рамок игры «Что? Где? Когда?». Доступность информации снизила ее значимость, ценность стали представлять не сами сведения из статей энциклопедии, а владение технологиями.

Технологии в аппаратном обеспечении, «железе», подчинены законам физики, что делает их развитие предсказуемым с достаточной долей достоверности. В противоположность этому, мир программных технологий основан на математических и лингвистических моделях и подчинён законам ведения бизнеса. Возможность собирать изделия из «кубиков» стала предметом зависти софтостроителей, вылившейся в итоге в компонентный подход к разработке. Панацеи, разумеется, не получилось, несмотря на серьёзный вклад технологии в повторное использование «кубиков», оказавшихся скорее серыми ящиками с малопонятной начинкой.

Безысходное программирование – это программирование без «исходников». То есть мы пишем свой код, не имея исходных текстов используемой подпрограммы, класса, компонента и т. п. Когда необходимо обеспечить гарантированную работу приложения, включающего в себя сторонние библиотеки или компоненты, то, не имея доступа к их исходному коду, вы остаётесь один на один с «чёрным ящиком». Даже покрыв их тестами, близкими к параноидальным, вы не сможете понять всю внутреннюю логику работы и предусмотреть адекватную реакцию системы на нестандартные ситуации. Поэтому программирование без исходников в таком сценарии превращается в настоящую безысходность и безнадёгу.

Спустя 20 с лишним лет, все мы – и разработчики, и пользователи – продолжаем сидеть на «числогрызах» 4-го поколения. Производительность «железа» возросла на порядки, почти упёршись в физические ограничения миниатюризации полупроводников и скорость света. Стоимость тоже на порядки, но снизилась. Увеличилась надёжность, развилась инфраструктура, особенно сетевая. Параллелизация вычислений пошла в массы на плечах многоядерных процессоров. Возросла ли при этом скорость разработки? Вопрос достаточно сложный, даже если сузить периметр до программирования согласно постановке задачи. Тем не менее, можно утверждать, что не только не возросла, но, наоборот, снизилась.

Идея универсального программируемого терминала, которым является веб-браузер, поддерживающий стандарты взаимодействия с веб-сервером, выглядит привлекательно. Никакого развёртывания, никакого администрирования на рабочем месте. Именно этот аргумент и стал решающим в конце 1990-х годов для внедрения веб в корпоративную среду. Гладко было на бумаге, но забыли про овраги…

К началу 2000-х годов установился фактический стандарт корпоративного веб-приложения: Internet Explorer 6 с последним пакетом обновления под Windows 2000 или Windows XP.

Появившись в 1995 году, технология Java сразу пошла на штурм рабочих мест и персональных компьютеров пользователей в локальных и глобальных сетях. Наступление проводилось в двух направлениях: полноценные «настольные» (desktop) приложения и так называемые апплеты, то есть приложения, имеющие ограничения среды исполнения типа «песочница» (sandbox). Например, апплет не мог обращаться к дискам компьютера.

Несмотря на значительные маркетинговые усилия корпорации Sun, результаты к концу 1990-х годов оказались неутешительны: на основной платформе пользователей – персональных компьютерах – среда исполнения Java была редким гостем. Основными объективными причинами такого исхода были: • универсальность и кроссплатформенность среды, обернувшаяся низким быстродействием и невыразительными средствами отображения под вполне конкретной и основной для пользователя операционной системой Windows; • необходимость установки и обновления среды времени исполнения (Java runtime).

Ниша к началу 2000-х годов оказалась плотно занятой Flash-приложениями, специализирующимися на отображении мультимедийного содержания. Учтя ошибки Java, разработчики из Macromedia сделали инсталляцию среды исполнения максимально лёгкой в загрузке и простой в установке.

Побочным продуктом WPF стал Silverlight. По сути, это реинкарнация Java-апплетов, но в 2007 году, спустя более 10 лет, и в среде. NET. Кроме того, Silverlight должен был по замыслу авторов составить конкуренцию Flash в области мультимедийных интернет-приложений. Для развёртывания не требовалась вся среда .NET целиком, достаточно было установить её часть, размер дистрибутива которой составлял всего порядка 5 мегабайтов.

Silverlight вырос до вполне взрослой версии 4, уже давно вышла Visual Studio 2010, где встроена поддержка разработки приложений под него. Но зададимся вопросом: «Может ли пользователь

установить себе Silverlight-приложение, не будучи администратором на своем компьютере?» Ответ, мягко говоря, разочаровывающий: «Нет, не может».

Из продвигаемых Microsoft за последние 10 лет технологий для разработки полноценных пользовательских интерфейсов, не заброшенных на пыльный чердак, остался только WPF, имеющий весьма сомнительную ценность для небольших коллективов и отдельных разработчиков.

Говоря об объектно-ориентированном подходе и программировании, принято добрым словом вспоминать начало 1970-х годов и язык Smalltalk, скромно умалчивая, что понадобилось ещё почти 15 лет до начала массового применения технологии в отрасли, прежде всего, за счёт появления C++ и позднее – Объектного Паскаля.

Как только «главным программистом» стал «коллективный разум» муравейника, проблема мгновенно всплыла, порождая Ад Паттернов, Чистилище нескончаемого рефакторинга и модульных тестов, недосягаемый Рай генерации по моделям кода безлюдного Ада.

Термин «Ад Паттернов» может показаться вам незнакомым, поэтому его следует расшифровать подробнее: • 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 сложных и многоуровневых конструкций в стиле «новый адаптер вызывает старый» по мере так называемого эволюционного развития системы; • лоскутная интеграция существующих систем и создание поверх них новых слоёв API.

Код начинает изобиловать плохо читаемыми и небезопасными конструкциями: Services.Oragnization.ContainerProvider.ProviderInventory.InventorySectorPrivate. Stacks[0].Code.Equals("S01"). Последствия от создания Ада Паттернов ужасны не столько невозможностью быстро разобраться в чужом коде, сколько наличием многих неявных зависимостей.

Несомненно, C++ является мощным инструментом программиста, хотя и с достаточно высоким порогом входа, предоставляющим практически неограниченные возможности профессионалам с потребностью технического творчества.

Когда многие технические проблемы были решены, оказалось, что ООП очень требовательно к проектированию, так и оставшемуся сложным и недостаточно формализуемым процессом. Похожая ситуация была в середине XX века в медицине: после изобретения антибиотиков первое место по смертности перешло от инфекционных болезней к сердечно-сосудистым.

Реальную отдачу от ООП вы получите, только создав достаточно хорошие модели предметных областей. То есть минимально необходимый словарь и язык вашей системы для выражения её потребностей. Модели будут настолько простыми и ясными, что их реализация не погрузит команду в инфернальные нагромождения шаблонных конструкций и непрерывный рефакторинг, а фреймворки будут легко использоваться прикладными программистами без помощи их авторов. Правда, тогда неизбежно встанет вопрос о необходимости использования ООП…

Чем занимались в СССР в 1972 году проектировщики, техники и организаторы? Создавали промышленные вычислительные системы, отраслевые стандарты и их реализации. Чтобы легче было собирать статистическую информацию. Чтобы снижать издержки на инфраструктуру, оборудование и эксплуатацию. ЭВМ Единой Серии, относились к классу больших универсальных ЭВМ. Если частично заменить «большие ЭВМ» на «кластеры из серверов», добавить взращенную широкую полосу пропускания общедоступных сетей на «последнем километре», а в качестве терминала использовать веб-браузер или специальное приложение, работающее на любом персональном вычислительном устройстве, от настольного компьютера до коммуникатора, то суть не изменится. Облако – оно же ВЦКП для корпоративного и даже массового рынка. В чем состояли просчёты советской программы

ВЦКП? Их два. Один крупный: проглядели стремительную миниатюризацию и, как следствие, появившееся обилие терминалов. Второй: просчитались по мелочам, не спрогнозировав развал страны с последующим переходом к состоянию технологической зависимости.

Программировать распределённое приложение сложнее, чем централизованное. Не только технически, но и организационно. Но в качестве выигрыша получаем автономию сотрудников, рабочего места. Наиболее очевидное применение децентрализации – автономные программы аналитической обработки данных. Мощность терминала позволяет хранить и обрабатывать локальную копию части общей базы данных, используя собственные ресурсы и не заставляя центральный сервер накаляться от множества параллельных тяжёлых запросов к СУБД.

Услуга от продукта, материального товара, принципиально отличается минимум по трём пунктам:

- услуги физически неосязаемы;
- услуги не поддаются хранению;
- оказание и потребление услуг, как правило, совпадают по времени и месту.

CORBA73 – одна из технологий создания сервис-ориентированной архитектуры. CORBA имела очевидные достоинства, прежде всего это интероперабельность и отраслевая стандартизация не только протоколов (шины), но и самих служб и общих механизмов. Однажды реализованный программистами сервис мог использоваться многими системами без какой-либо дополнительной адаптации и ограничений с их стороны.

Цифры версий и релизов фреймворка .NET меняются со скоростью, заметно превышающей сроки отдачи от освоения и внедрения технологий. Давайте подумаем, кто же выигрывает в этой гонке кроме самой корпорации, пользующейся своим квазимонопольным положением:

- Услуги по сертификации. Прямая выгода.
- Консультанты и преподаватели курсов. Сочетание прямой выгоды с некоторыми убытками за счёт переобучения и очередной сертификации.
- Программисты в целом. Состояние неопределённости. Выбор стоит между «изучать новые возможности» и «решать задачи заказчиков».
- Разработчики в заказных проектах. Прямые убытки. За пару лет получен опыт работы с технологиями, признанными в новом фреймворке наследуемыми, то есть не подлежащими развитию, хорошо, если поддерживаемыми на уровне исправления критичных ошибок. Необходимы новые инвестиции в обучение персонала и преодоление появившихся рисков.
- Разработчики продуктов. Косвенные убытки. В предлагаемом рынку продукте важна функциональность и последующая стоимость владения. На чём он написан – личное дело компании-разработчика.

Разбираться в слоях и уровнях должны не только разработчики КИС, то есть «скелета», но и те программисты, которые будут наращивать на него свои приложения.

Определение автоматизированной информационной системы (АИС) складывается из трёх основных её компонентов: людей, информации и компьютеров. Любая АИС – это люди, использующие информационную технологию средствами автоматизации.

Любая АИС может быть рассмотрена с трёх точек зрения проектировщика:

- концептуальное устройство (существуют в любой системе);
    - отображение данных
    - обработка данных
    - хранение данных

- логическое устройство (слои неопределенные);
    - отображение данных
        - вид
        - контроллер
        - модель
    - обработка данных
        - расчеты
        - бизнес-правила
        - доступ к данным
    - хранение данных
        - API
        - обработка запросов
        - структуры
- физическое устройство (слои определяются на этапе проектирования реализации).
    - СУБД
    - тонкий клиент
    - толстый клиент
    - браузер
    - сервер приложений

Концептуальных слоёв в автоматизированной информационной системе всегда три: хранения данных, их обработки и отображения. А вот физических, их реализующих, может быть от одного, в виде настольного приложения с индексированным файловым хранилищем, до теоретической бесконечности.

Запустив в эксплуатацию и получив в сопровождение систему, программисты на собственном опыте убедились, что каждый физический слой увеличивает трудоёмкость разработки, тестирования и последующих модификаций системы. Поэтому, с учётом предполагаемого тиражирования, развивать продукт было решено в следующих направлениях:

- минимизация числа звеньев;
- «утончение» клиентского приложения, в идеале до уровня веб-браузера;
- использование промышленной СУБД для реализации бизнес-логики;
- реализация некоторых механизмов ООП для упрощения разработки прикладными и сторонними разработчиками методом надстраивания новых классов.

Синтезом вышеназванных приоритетов явилась двухзвенная архитектура с тонким клиентом.

Проектированием системы должен заниматься не технарь, а маркетолог. Маркетолог может понять потребности рынка, сформулировать требования к товару, может оценить осуществимость маркетинговой компании. На деле технический архитектор нужен маркетологу для оценки себестоимости проекта и его сроков, а также для информации о новых перспективных технологиях, тогда маркетолог сможет искать сочетания «потребность + технология».

Шаблоны сыграли с программистским сообществом злую шутку: они оказались зависимыми от языка программирования. То есть, прочитав «банду четырёх», писавших свой труд исходя из возможностей C ++, срочно бегите за новой порцией информации, кто для Java, кто для C#. Хотя задумывались шаблоны с обобщающей практики целью.

Думать надо не шаблонами, а головой.

Откажитесь от термина «наследование», который искажает смысл действий. Мы обобщаем. Обобщение (наследование) реализации или интерфейсов – весьма неоднозначный механизм в объектно-ориентированном программировании, его применение требует осторожности и обоснования.

Основное правило: обобщаемые классы должны иметь сходную основную функциональность. Основная ошибка – обобщение по неосновному функциональному признаку. Эмпирическое правило: глубина более двух уровней при моделировании объектов предметной области, вероятнее всего, свидетельствует об ошибках проектирования.

Сборщик мусора, он же GC – garbage collector в средах программирования с автоматическим управлением памятью. Наиболее очевидное преимущество – программисту не надо заботиться об освобождении памяти. Хотя при этом все равно нужно думать об освобождении других ресурсов, но сборщик опускает планку требуемой квалификации и тем самым повышает массовость использования среды. Чтобы избежать многих проблем нужно взять за правило, что контейнер всегда управляет своими объектами. Поэтому обращаться к его внутренним объектам нужно только через интерфейс самого контейнера.

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.

Всякий раз разработчики очередного корпоративного приложения создают свою «птолемеевскую систему», которая при попытке примерить ее на другую фигуру внезапно оказывается неподходящей и нуждается в серьёзной перекройке. Так и земная птолемеевская система не работает для наблюдателя, находящегося на Марсе.

Большой интерес представляют так называемые «двусторонние» инструменты (two-way tools), тесно интегрированные со средами программирования и позволяющие непосредственно во время разработки оперировать диаграммами с одновременным отражением изменений в коде и, наоборот, импортировать меняющийся руками код в модель. Однако при коллективной работе программистов достоинства этих инструментов сильно уменьшаются.

В итоге: UML, за неимением адекватных альтернатив в формализовании и фиксировании большого потока иногда противоречивых требований, является довольно неплохим инструментом для генерации кода. Некоторые диаграммы в UML описываются максимально неконкретно, что может повлечь совершенно разную реализацию одного и того же 2 разными людьми, что, на мой взгляд, является минусом.

Построив распределённую архитектуру клиент-серверного приложения, мы сталкиваемся с двумя основными проблемами:

- поддержка данных в актуальном состоянии;
- обработка данных относительно слабым, по сравнению с сервером, компьютером пользователя.

Нужно учитывать, что на клиенте не может находиться большая часть БД, что накладывает определенные ограничения. Однако при грамотной реализации производительность таких приложений, по сравнению с приложениями с БД, находящейся только на сервере, ее

производительность гораздо выше, что может являться критичным при работе с большими объемами данных.

При неграмотной же разработке приложения может случиться, что чрезмерная абстракция и излишняя фильтрация информации значительно замедлит работу приложения, что даже создание «костылей» не спасет систему.

Вывод можно извлечь такой: нужно понимать работу приложения не только в общих чертах, но зачастую и на физическом уровне, что даст возможность более гибкой оптимизации ее работы.

Один из «Технических Дней Microsoft» (TechDays) в 2011 году был целиком посвящён специализации DBA (DataBase Administrator). А выступающий на сцене ведущий эксперт не постеснялся напрямую высказать призыв: «Последние годы я вижу тотальное падение компетенции в области баз данных. DBA, проснитесь!»

Ревизия кода весьма полезная процедура, но как минимум при двух условиях:

- эта процедура регулярная и запускается с момента написания самых первых тысяч строк;
- процедуру проводят специалисты, имеющие представление о системе в целом. Потому что отловить бесполезную цепочку условных переходов может и компилятор, а вот как отсутствие контекста транзакции в обработке повлияет на результат, определит только опытный программист.

Принцип для заказчика: «Быстро, качественно, дёшево – выбери два критерия из трёх».

Методы разработки:

- «Водопад» также соответствует разработке «сверху-вниз» в структурном программировании: выделяем самые общие функции системы, проводим их декомпозицию на подфункции, а те, в свою очередь, на подподфункции и так далее, пока не упрёмся в элементарные операции.
- Спираль. Ключевой особенностью спиральной технологии является прототипирование. В конце каждого витка после этапа стабилизации заказчик получает в своё распоряжение ограниченно работающий прототип целой системы, а не отдельных функций. Основная цель прототипа состоит в максимально возможном сближении взглядов заказчика и подрядчика на систему в целом и выявлении противоречивых требований.
- Гибкая модель. Ключевой особенностью гибкой методики является наличие мифологического титана – владельца продукта (product owner), который лучше всех знает, что должно получиться в итоге. Именно владелец, за рамками собственно гибкого процесса, проводит анализ и функциональное проектирование, подавая команде на вход уже готовые пачки требований. Размер пачки должен укладываться в интеллектуальные и технологические возможности разработчиков, которым предстоит осуществить её реализацию за одну итерацию.

Разработка модульных тестов – это разработка. Для 100 % покрытия потребуется примерно столько же времени, сколько и на основную работу. А может, и больше.

Модульные тесты 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.

Соизмеряйте затраты на создание и поддержу автоматизированных модульных тестов с бюджетом проекта и располагаемым временем.

Идея разрабатывать программы, минимизируя стадию кодирования на конкретных языках под заданные платформы, появилась достаточно давно.

В качестве решения многих проблем появились так называемые двусторонние CASE-инструменты (two way tools), позволяющие редактировать как модель, непосредственно видя изменения в коде, так и, наоборот, менять код с полу– или полностью автоматической синхронизацией модели. Зачастую, такой инструмент был интегрирован прямо в среду разработки.

Программист описывает задачу в терминах логической модели, представляющей собой набор сущностей, их атрибутов, операций и связей между ними.

Соотношение числа строк мета-кода описания модели к коду его реализации на конкретных архитектурах и платформах составляет около 1 к 50.

Это означает, что оснащённый средствами автоматизации программист с навыками моделирования на этапе разработки рутинного и специфичного для платформ/архитектур кода производителен примерно так же, как и его 50 коллег, не владеющих технологией генерации кода по моделям. Любое внесение изменений в модель тут же приводит в соответствие все генерируемые слои системы, что ещё более увеличивает разрыв по сравнению с ручными модификациями.

Наконец, для генерируемого кода не нужны тесты. Производительность возрастает ещё как минимум вдвое.

Ипатов Алексей

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

Появилась огромная масса проектов, некритичных к срокам и качеству выполнения. При этом граница применения компьютеров расширилась до областей, казавшихся ранее недоступными.

Исторически сложилось так, что многие программисты были преимущественно математиками и самостоятельно занимались формализацией задач. То есть приведением их к виду, пригодному для решения на компьютере. Сам себе постановщик и кодировщик. В общем виде формула, отражавшая суть работы выражалась так:

Программист = алгоритмизация и кодирование

С ростом сложности задач и упрощения процесса непосредственного кодирования при одновременном усложнении повторного использования кода появилась возможность разделить труд, и формула приобрела примерно такой вид:

Программист минус алгоритмизация = кодировщик

Программист минус кодирование = постановщик задачи

Это вовсе не значит, что мир разделился на аналитиков-алгоритмистов и техников-кодировщиков. Практика многих десятилетий показала, что по-прежнему наиболее востребованным специалистом является инженер, способный как самостоятельно формализовать задачу, так и воспользоваться стандартны

ми средствами её решения на ЭВМ.

Иерархия подразделений

(*) – отдел является единицей финансирования, то есть бюджеты составляются, начиная с уровня отдела;

(**) – уровень лаборатории не являлся обязательным для организаций, не ведущих НИР (научно-исследовательскую работу), например, для ВЦКП (Вычислительный Центр Коллективного Пользования).

Иерархия должностей

Уровни принятия проектных решений

Уровни инвариантны организации, они существуют всегда, но могут быть размытыми, например, если проект небольшой. Используется иерархия деления: система – подсистема – модуль.

Функциональные специализации (роли)

(*) – как правило, главный конструктор или ГИП занимали должности от начальника сектора и выше в зависимости от проекта, который они возглавляли;

(**) – уровень архитектора подсистемы/направления соответствует уровню ведущего инженера. Направления специализации могут быть разнообразными: базы данных, человеко-машинный интерфейс, качество (испытания), информационная без

опасность, сетевое оборудование, инфраструктура и т. д.

Метаморфозы

В штате институтского ВЦ или конструкторско-технологического центра всегда присутствовали «инженер-программист» и «техник-программист».

Формальное различие состояло в том, что техников готовили в профильных профессионально-технических училищах (ПТУ) и техникумах; были математики-программисты, которых готовили в университетах.

Фактическое же различие состояло в том, что техники не занимались постановкой задач и проектированием программных систем, ограничиваясь непосредственно программированием и эксплуатацией.

Системный программист-техник большой ЭВМ превратился в системного администратора по эксплуатации сети «персоналок» и серверов.

Современная ситуация изменилась, нередко диплом о высшем образовании, ранее гарантировавший уровень, достаточный для допуска инженера к проектированию, на практике подтверждает лишь уровень техника.

Случается наблюдать, как новоиспечённый системой высшего образования программист утверждает: «Мне не понадобилось ничего из того, чем пичкали в институте». Но если эта фраза, произнесенная с

гордостью, означает, что работа никчёмная, то эта же фраза, произнесённая с грустью, говорит о том, что вуз – бесполезный.

Софтостроение можно рассмотреть, с одной стороны, как средство заработка на жизнь, а с другой – как средство реализации своих идей в техническом творчестве. Сделать не просто «чтобы работало», а чтобы ещё и «было красиво».

Любить себя в софтостроении, а не софтостроение в себе.

Разумеется, для таких работников главной, а то и единственной целью будет сделать «чтобы как-то работало». Это даже не ремесло в чистом виде, а неизбежные плоды попыток индустриализации в виде халтуры и брака.

В итоге программирование нельзя целиком причислить ни к искусству, ни к ремеслу, ни к науке. Софтостроение на текущий момент – эклектичный сплав технологий, которые могут быть использованы как профессионалами технического творчества, так и профессионалами массового производства по шаблонам и прецедентам. Кадры решат всё.

Производительность труда растёт, высвобождающиеся из производственных цепочек люди вынуждены уходить в сферу услуг. Но и она не бездонна

В Германии совершенно серьёзно и открыто обсуждается идея выплаты всем гражданам безусловного основного дохода9 (БОД) – минимального пособия примерно в 1000 евро, которого хватит на оплату недорого жилья, скромного питания и одежды

Имея возможность потреблять, не отдавая взамен свой труд, останетесь ли вы профессионалом в своей сфере?

На одного производителя тиражируемого продукта разной степени серийности – от массовых брендов до малотиражных специализированных, приходится почти десяток поставщиков услуг, крутящихся вокруг этих продуктов, и разработчиков заказных программных систем.

Уникальность софтостроения как сферы услуг в его высокой кадровой ёмкости.

Тем временем уволенные с основного производства приходят в службу занятости, где им говорят: «Специалистов вашего профиля повсюду сокращают. Предлагаем вам переквалификацию». И

дружными рядами бывшие операторы устаревшей автоматизированной линии идут на трёхмесячные курсы «Разработка приложений в среде Basic» или «Разработка веб-приложений». После окончания учёбы они попадают на работу в фирму-подрядчик, где начинают автоматизировать использование электронных таблиц отделом компании, откуда их несколько месяцев назад сократили.

Непосредственная власть находится в руках управленцев среднего звена.

На практике замена старого подрядчика новым – весьма рискованная процедура даже на некритичных

проектах. Менеджеры стремятся, с одной стороны, затраты, наоборот, сократить, а с другой – максимально раздуть бюджет и штат для его освоения ради продвижения по карьерной лестнице.

В софтостроении такая оптимизация оказывается проблематичной. Имеет место и другая веская причина – нечёткость требований, сформулировать которые заказчик далеко не всегда в состоянии. У подрядчика же функциональная специализация программистов существует только для клиентов, способных давать стабильный заказ с высокой долей прибыли.

Мельница крутится, в разработку «проектов для отделов «Х» и следующую за этим через несколько лет переделку втягивается всё больше людей.

Когда вступают в действие большие числа, впору вспомнить о нормальном распределении, на которое нам открыл глаза ещё старина Гаусс.

Рис. 1. Нормальное распределение уровня профессиональной компетентности программистов

Чтобы не просто зарабатывать на хлеб, но и мазать его маслом, сохраняя при этом возможности технического творчества, вам лучше держаться подальше от тех направлений деятельности, где конкурентами будут 6 миллионов человек.

Я не верблюд, чтобы доказывать, что я не верблюд.

Количество проектов, где потребуется ваша квалификация, намного меньше количества некритичных заказов, а большинство ваших попыток проявить свои знания и умения столкнется с нелояльной конкуренцией со стороны вчерашних выпускников курсов профессиональной переориентации. Придётся снижать цену своего труда и готовиться к менее квалифицированной работе.

Новые значимые проекты возникают только с новыми рынками и направлениями бизнеса.

Поэтому немало специалистов высокой квалификации уходят в экспертизу и консалтинг, где проводят аудит, обучение, «натаскивание» и эпизодически «вправляют мозги» разным группам разработчиков из числа переквалифицировавшихся.

Другой доступный вариант – специализация на предметных областях. В этом случае разработчик относительно автономен и, во-первых, гораздо менее ограничен в выборе инструментов. Во-вторых, что более существенно, доказывать кому-то степень владения инструментарием у него нет необходимости. Обычная эволюция такого специалиста

– системный аналитик, сохранивший знания технологий времён своего последнего сеанса кодирования в интегрированной среде.

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

Взаимодействия типа «человек – человек» становятся необходимым и важным элементом повседневной работы, если только вы не предполагаете всю жизнь провести в кодировании чужих спецификаций, не всегда толковых и формализованных. Вместо решения сугубо технических задач

вашей целью будет решение задач конкретных клиентов. А критерием решения станет субъективная степень удовлетворённости клиента.

Малозначимым в глазах руководства может оказаться не только проект, но и обслуживание крупного продукта

Проблема, удовлетворительных решений которой на сегодняшний день не найдено. Поэтому и обсуждают введение пособий типа БОД: пусть лучше получают свой минимум и занимаются чем хотят, чем портят отношения с клиентами, мешая производительному труду остальных.

Небольшой словарик ключевых фраз, часто присутствующих в объявлении о вакансии

1. «Быстро растущая компания» – фирма наконец получила заказ на нормальные деньги.

2. «Гибкие (agile) методики» – в конторе никто не разбирается в предметной области на системном уровне.

3. «Умение работать в команде» – в бригаде никто ни за что не отвечает, документация потеряна или отсутствует с самого начала.

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.

5. «Гибкий график работы» – программировать придётся «отсюда и до обеда».

6. «Опыт работы с заказчиком» – заказчик точно не знает, чего хочет, а зачастую – неадекватен в общении.

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.

Тесты – особый пункт при найме. Чаще всего они касаются кодирования

Не стоит мерить интеллект тестом на IQ15. Лучше давать испытуемому некоторый нестандартный тест, чтобы просто посмотреть на ход его мысли. Чарльза Уэзерелла «Этюды для программистов» [17].

Резюме, является важной деталью вашего представления потенциальному работодателю.

принципы хорошего резюме следующим образом:

- Краткость – сестра таланта. Поэтому постарайтесь на первой странице поместить всю основную информацию: ФИО, координаты, возраст, семейное положение, мобильность, личный сайт или блог, описания своего профиля, цель соискания, основные технологии с оценкой степени владения (от «применял» до «эксперт»), образование, в том числе дополнительное, владение иностранными языками.
- Кто ясно мыслит, тот ясно излагает. Все формулировки должны быть ёмкими и краткими.
- Не фантазируйте.
- Тем более не врите

Одного прокола будет достаточно для попадания в «чёрный список» компании.

- В каждом описании опыта работы упирайтесь на технологии.
- Не делайте ошибок

• Будьте готовы, что далее первой страницы ваше резюме читать не станут.

Соискатель: Вы используете так называемые «гибкие» методы, например, Scrum? Если да, то какова степень формализации процесса? У вас есть аналитики и проектировщики? Какие модели вы используете? Есть ли практика ежедневных утренних планёрок? Есть ли ответственные за подсистемы?

Работодатель: Да – высокая – выделенных нет – что-то рисуется в UML – обязательно! – есть, трудовой коллектив.

Соискатель: Спасибо, всего вам доброго и успехов в труде!

Про мотивацию

Около дома одного человека мальчишки играли в мяч: ударяли им о стены, громко кричали

и смеялись. Естественно, они мешали хозяину дома. И вот в один прекрасный день он вышел к ним и сказал: «Друзья, вы так весело играете в мяч, так заразительно смеётесь и кричите, что я с удовольствием вспоминаю свое детство. Я буду платить каждому по монете, чтобы вы каждый день приходили сюда, громко кричали, смеялись

и играли в мяч». Мальчишки взяли по монете и продолжили игру. На следующий день они снова пришли и получили по монете. Так продолжалось несколько дней. Но как-то хозяин подошёл к мальчишкам и сказал, что его финансовые дела не так хороши, как раньше, и он сможет платить им только по полмонеты. Он заплатил им по полмонеты и ушёл. А мальчишки поговорили и решили, что не будут стараться за полмонеты. И больше они не приходили. Так хозяин дома получил желаемые мир и спокойствие.

Мотивация – исключительно внутренний механизм. Чтобы управлять им, необходимо залезать в этот самый механизм, в психику. Существуют традиционные способы: педагогика и воспитание. Они требуют многих лет, а то и смены поколений. Существуют и более быстрые варианты, связанные с химией и

Напротив, стимуляция – это внешний механизм. Он основан на выявлении мотивов и последующем их поощрении или подавлении. Стимулятор подобен катализатору для запуска химической реакции. Управление стимуляцией сводится к созданию системы стимулов, требуемых для «реакций» катализаторов.

. Мотивировать же можно только свои поступки, но никак не образ действия окружающих.

Изгибы судьбы при поиске работы

Достаточно разместить резюме на одном из крупных вебсайтов, параллельно входя напрямую в контакт с отдельными компаниями.

Второй этап – выяснение, ищут ли они специалиста на конкретный проект или просто под широкий профиль.

Третий этап – отбрыкаться от приглашения на бесполезное интервью, убедив, что не стоит зря тратить время. Ведь у мадемуазелек рабочего времени навалом.

Коллеги, не будьте холодны с мадемуазельками-ассистентками из кадровых служб!

На мой взгляд, наиболее полезной была информация о грамотном составлении резюме, поиске работы и правильном общении с “ мадемуазельками”.

Не согласен я с идеей выплаты всем гражданам Германии или другого государства безусловного основного дохода (БОД).

Доступность информации снизила ее значимость, ценность стали представлять не сами сведения из статей энциклопедии, а владение технологиями.

Технологии в аппаратном обеспечении, «железе», подчинены законам физики, что делает их развитие предсказуемым с достаточной долей достоверности. Через десятилетие мир вычислительных устройств изменится.

Радикальных изменений в софтостроительной сфере ожидать не следует, ситуация находится под чутким контролем крупных корпораций и развивается эволюционно.

Тем не менее в софтостроении, даже кустарном и далёком от индустриализации, технологии составляют основу.

Основой индустриализации в производстве «железа» стали проектирование и сборка устройств из стандартизованных компонентов.

В софтостроении тоже имеются относительно стандартные подсистемы: операционные среды, базы данных, веб-серверы, программируемые терминалы и тому подобное.

Для аппаратуры используется модель конечного автомата. Во-первых, она обеспечивает полноту тестирования. Во-вторых, компонент работает с заданной тактовой частотой, то есть обеспечивает на выходе сигнал за определённый интервал времени. В-третьих, внешних характеристик (состояний) у микро

схемы примерно два в степени количества «ножек»,

что на порядки меньше, чем у программных «кубиков». В-четвёртых, высокая степень стандартизации даёт возможность заменить компоненты одного производителя на другие, избежав сколько-нибудь значительных модификаций проекта.

В софтостроении использовать конечно-автоматную модель для программного компонента можно при двух основных условиях:

- Программисту не забыли объяснить эту теорию ещё в вузе (см. выше про «Круговорот»).
- Количество состояний обозримо: они, как и переходы, достаточно легко определяются и формализуются.

Поэтому вдобавок к модульному тесту необходимо программировать тест производительности (нагрузочный), который тем не менее не гарантирует время отклика, а только позволяет определить его ожидаемое значение при некоторых условиях.

Таким образом, собрав из кучи микросхем устройство, мы уверены, что оно будет работать:

- согласно таблицам истинности;
- с заданной тактовой частотой.

Собрав же из компонентов программу, мы можем только:

- приблизительно и с некоторой вероятностью оценивать время отклика на выходе;
- в большинстве случаев ограничиться выборочным тестированием, забыв о полноте.

Безысходное программирование – это программирование без «исходников». То есть мы пишем свой код, не имея исходных текстов используемой подпрограммы, класса, компонента и т. п.

Когда необходимо обеспечить гарантированную работу приложения, включающего в себя сторонние библиотеки или компоненты, то, не имея доступа к их исходному коду, вы остаётесь один на один с

«чёрным ящиком». Даже покрыв их тестами, близкими к параноидальным, вы не сможете понять всю внутреннюю логику работы и предусмотреть адекватную реакцию системы на нестандартные ситуации, а ответственность с разработчиков никто не снимал.

Частным, но частым случаем безысходного программирования является софтостроение без использования системы управления исходным кодом (revision control system), позволяющей архивировать и отслеживать все его изменения.

Грабли – синоним скрытой ошибки в программе. «Наступить на грабли» в программистском фольклоре означает выявить скрытую проблему за собственный счёт.

Эволюция аппаратуры

и скорость разработки

В 1980х годах у японцев существовала программа по созданию ЭВМ 5го поколения. К сожалению, цель достигнута не была, хотя проявилось множество побочных эффектов вроде всплеска интереса к искусственному интеллекту, популяризации языка Пролог, да и отрицательный опыт – тоже опыт, возможно, не менее ценный.

В итоге, спустя 20 с лишним лет, все мы – и разработчики, и пользователи – продолжаем сидеть на «числогрызах» 4го поколения. Производительность «железа» возросла на порядки, почти упёршись в физические ограничения миниатюризации полупроводников и скорость света. Стоимость тоже на порядки, но снизилась. Увеличилась надёжность, развилась инфраструктура, особенно сетевая. Параллелизация вычислений пошла в массы на плечах многоядерных процессоров.

Прежними остались лишь принципы, заложенные ещё в 1930х годах и названные, согласно месту, Гарвардской и Принстонской архитектурами ЭВМ.

Возросла ли при этом скорость разработки? Вопрос достаточно сложный, даже если сузить периметр до программирования согласно постановке задачи. Тем не менее я рискнул бы утверждать, что не только не возросла, но, наоборот, снизилась.

Специализированные средства разработки всегда обеспечат преимущества по сравнению с универсальными. Тем не менее основная разработка по-прежнему будет идти на весьма ограниченном наборе универсальных сред и фреймворков, выталкивая специализированную в ниши, где сроки и производительность являются наиболее важными.

Количество работающих в софтостроении женщин росло до начала 1990х годов, после чего резко пошло на убыль

Такая тенденция иллюстрирует факт ухода технологий софтостроения от специализированных сред, не требующих работы на далёком от решаемой прикладной задачи уровне математических абстракций, в которых прекрасный пол почему-то считается менее способным разбираться.

Про объектно-ориентированный подход мы ещё поговорим, но у меня сложилось мнение, что будучи реализованной повсеместно без малейших представлений о её применимости, технология ООП сыграла не последнюю роль в вытеснении женского труда из отрасли.

На мой взгляд, наиболее необычной и полезной была информация о безысходном программировании. Честно говоря, никогда не слышал. Так же интересна тема ЭВМ нового поколения, но, к сожалению, об этом есть лишь какое-то упоминание.

Не согласен я тем, количество женщин работающих в софтостроении постепенно сокращается. Я как-то случайно услышал, что наоборот большинство компаний хвастаются , что стали нанимать на работу больше женщин, афроамериканцев и латинос.

Механизм принятия решения базировался на других критериях:

- менеджеру, в соответствии с корпоративным стандартом, необходимо было использовать только платформы и средства Microsoft;
- программист не имел опыта разработки вне шаблонов многозвенной архитектуры и проекций объектов на реляционную СУБД, поэтому не стал рисковать.

Такие решения принимаются в мире ежечасно. Поэтому новичкам не раз предстоит столкнуться с заданием типа «быстро добавить поле в форму» и познакомиться с внутренним устройством подобных программ – карманных монстров, готовых откусить палец неосторожно сунутой руки.

веб быстрым темпом трансформируется из источника статической информации в среду динамических интерактивных приложений, доступных через «стандартный» проводник-браузер Microsoft не могла остаться в стороне и выдала собственное решение под названием ASP (Active Server Pages), работающее, разумеется, только под Windows.

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.

В итоге исходная веб-страница, ранее содержавшая только разметку гипертекста, стала включать в себя скрипты для выполнения вначале на сервере, а затем и на клиенте.

Последующая эволюция технологии была посвящена борьбе с этой лапшой, чтобы программный код мог развиваться и поддерживаться в большем объёме и не только его непосредственными авторами. На другом фронте бои шли за отделение данных от их представления на страницах, чтобы красивую обёртку рисовали профессиональные дизайнеры графики, не являющиеся программистами.

несмотря на значительный прогресс за последние 15 лет, производительность разработки пользовательского интерфейса для веб-приложений в разы отстаёт от автономных приложений

несмотря на то, что создатели чётко отделили слой представлений от прикладной обработки, следовали логике «модель– представление – котроллер» 23, а общие элементы управления разного уровня – от собственных (custom) до композитных (user) – свели в библиотеки.

. Основной целью такой дополнительной головной боли является платформенная независимость клиентской части приложения и максимально облегчённое развёртывание так называемого «тонкого» клиента, которым является веб-браузер.

переносить автономное приложение между разными операционными системами и аппаратными платформами трудно.

Поэтому на первый взгляд идея универсального программируемого терминала, которым является веб-браузер, поддерживающий стандарты взаимодействия с веб-сервером, выглядит привлекательно. Никакого развёртывания, никакого администрирования на рабочем месте.

Достаточно быстро выяснилось, что разработка приложения, корректно работающего хотя бы под двумя типами браузеров (Internet Explorer, Netscape и впоследствии Mozilla) – задача не менее сложная,

А тестировать нужно не только под разными браузерами, но и под разными операционными системами. С учётом версий браузеров.

эту проблему решили в лоб. Корпоративная среда в отличие от общедоступного Интернета имеет свои стандарты. Поэтому при разработке веб-приложений достаточно было согласовать внутренние требования предприятия с возможностями разработчиков.

На деле же оказалось, что, собрав свои приложения на базе веб-технологии, корпорация оказалась заложником версии и марки конкретного браузера.

за 5–10 лет сменяются разработчики, уходят с рынка прежние поставщики. Для таких случаев приложение остаётся жить на виртуальной машине под старой версией операционной системы и проводника.

даже в самом примитивном варианте развёртывания при использовании обычных исполняемых файлов с запуском с разделяемого сетевого диска обновление одной программы не вызывает крах остальных.

ответственным за выбор технологий, всячески обосновывать необходимость использования веб-интерфейса в вашей системе, принимая в рассмотрение другие пути.

Появившись в 1995 году, технология Java сразу пошла на штурм рабочих мест и персональных компьютеров пользователей в локальных и глобальных сетях. Наступление проводилось в двух направлениях: полноценные «настольные» (desktop) приложения и так называемые апплеты24, то есть приложения, имеющие ограничения среды исполнения типа «песочница» (sandbox). Например, апплет не мог обращаться к дискам компьютера.

вебсайтов, поддерживавших апплеты, было исчезающе мало, а настойчивые просьбы с их страниц скачать и установить 20 мегабайтов исполняемого кода для просмотра информации выглядели издевательством при существовавших тогда скоростях и ограничениях трафика.

Но основными объективными причинами такого исхода были:

- универсальность и кроссплатформенность среды, обернувшаяся низким быстродействием и невыразительными средствами отображения под вполне конкретной и основной для пользователя операционной системой Windows;
- необходимость установки и обновления среды времени исполнения (Java runtime).

В развитие приложений на десктопах сместилось далеко в сторону по пути начинённых скриптами веб-браузеров, а рост количества мобильных устройств положил конец монополии «персоналок» в роли основного

пользовательского терминала.

Тем не менее необходимость в кроссплатформенных богатых интерактивными возможностями интернетприложениях27 никуда не исчезла, поскольку браузеры, нашпигованные скриптовой начинкой, обладали ещё большими техническими ограничениями и низким быстродействием даже по сравнению с апплетами.

Прежде всего насторожили меня новости про отсутствие в Silverlight отличных от юникода30 кодировок.

Прямой доступ к базам данных также отсутствовал.

Из продвигаемых Microsoft за последние 10 лет технологий для разработки полноценных пользовательских интерфейсов, не заброшенных на пыльный чердак, остался только WPF, имеющий весьма сомнительную ценность для небольших коллективов и отдельных разработчиков. WPF – это ниша крупных автономных Windows-приложений. Кроме того, сама по себе она невелика, в ней уже есть WinForms – более простой и быстрый в разработке фреймворк, к тому же переносимый под Linux/Mono. Поэтому при соответствующих ограничениях развёртывания выбор по-прежнему лежит между веб-браузером или условным Delphi.

На мой взгляд, наиболее необычной и полезной была информация о Silverlight и ASP. NET

Термин «Ад Паттернов» может показаться вам незнакомым, поэтому я расшифрую подробнее это широко распространившееся явление:

- 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 сложных и многоуровневых конструкций в стиле «новый адаптер вызывает старый» по мере так называемого эволюционного развития системы;
- лоскутная38 интеграция существующих систем и создание поверх них новых слоёв API39.

Поэтому появились новые Сподобные языки: сначала Java, а чуть позже и C#. Они резко снизили порог входа за счёт увеличения безопасности программирования, ранее связанной прежде всего с ручным управлением памятью. Среды времени исполнения Java и. NET решили проблему двоичной совместимости и повторного использования компонентов системы, написанных на разных языках для различных аппаратных платформ.

Когда многие технические проблемы были решены, оказалось, что ООП очень требовательно к проектированию, так и оставшемуся сложным и недостаточно формализуемым процессом

Но стоит применить такой подход к объектам реального мира, как возникнет необходимость во множественном наследовании от сотни разношёрстных абстрактных заготовок. Плодятся новые многоуровневые иерархии, но теперь уже не наследования (is a), а вложения (is a part of).

Дальше интерфейсы пересекаются, обобщаются, и мы получаем ту же самую иерархию наследования, от которой сбежали. Но теперь это уже иерархия, во-первых, множественная, а во-вторых, состоящая из абстрактных классов без какой-либо реализации вообще (интерфейс, по сути, есть pure abstract class).

Одна из причин подобных злоключений в том, что концепции, выдвигаемые ООП, на самом деле не являются его особенностями за исключением наследования реализации от обобщённых предков с виртуализацией их функций.

Наследование реализации является одним из основных механизмов порождения ада наследуемых ошибок, неявных зависимостей и хрупкого дизайна. Все остальные концепты от инкапсуляции и абстракции до полиморфизма имеются в вашем распоряжении без ООП.

Надо признать, входной порог использования ООП оказался гораздо выше, чем предполагалось в 1980– 90х гг.

Практика подтвердила, что технология, будучи обёрнутой в относительно безопасные языковые средства и жёсткие стандарты, допускает применение в больших проектах. С другой стороны, немалое число крупных проектов принципиально не используют ООП.

Реляционным СУБД удалось в 1980х годах освободить программистов от знания ненужных деталей организации физического хранения данных, отгородиться от них структурами логического уровня и стандартизованным языком SQL46 для доступа к информации. Также оказалось, что большинство форматов данных, которыми оперируют программы, хорошо ложатся на модель двумерных таблиц и связей между ними.

И реляционная и объектная модели относятся к логическому уровню48 проектирования программной системы.

И реляционная и объектная модели относятся к логическому уровню48 проектирования программной системы.

Это значит, что вы можете реализовать одну и ту же систему,

оставаясь в рамках только одного реляционно-процедурного подхода или же следуя исключительно ООП.

Говорят, что для слоя домена50 генерируемый проектором SQL код является аналогом результата трансляции языка высокого уровня в ассемблер целевого процессора. Это мнение не просто глубоко ошибочно, оно быстрыми темпами ведёт команду к созданию трудно-сопровождаемых систем с врождёнными и практически неисправимыми проблемами производительности.

Для эффективной работы с РСУБД нужно использовать подходы, ориентированные на обработку множеств на сервере, предполагающие наличие у разработчика умений работать с декларативными языками.

Несмотря на толстый слой абстракций, предоставляемый ORM типа Hibernate, заставить приложение эффективно работать с РСУБД без знаний соответствующих принципов ортогонального мира и языка SQL практически невозможно.

Дальше эволюция разработки системы примерно следующая:

•Вначале происходит выбор ORM-фреймворка для отображения.

•Начинаем реализовывать модель предметной области. Добавляем классы, свойства, связи. Генерируем структуру базы данных или подключаемся к существующей. Строим интерфейс управления объектами типа CRUD.

Внезапно оказывается, что собственный язык запросов генерирует далеко не самый оптимальный SQL. Когда БД относительно небольшая, сотня тысяч записей в наиболее длинных таблицах, а запросы не слишком сложны, то даже неоптимальный сиквел во многих случаях не вызовет явных проблем. Пользователь немного подождёт.

Тогда разработчики идут единственно возможным путём: выбираем коллекцию объектов и в циклах фильтруем и обсчитываем, вызывая методы связанных объектов.

Обычно разработчикам баз данных рекомендуют избегать необоснованного использования триггеров. Потому что их сложнее программировать и отлаживать. Оставаясь скрытыми в потоке управления, они напрямую влияют на производительность и могут давать неожиданные побочные эффекты. Пользуйтесь декларативной ссылочной целостностью (DRI58) и хранимыми процедурами, пока возможно.

СУБД – это достаточно простая и хорошо документированная технология. В то время как NHibernate предлагает прикладному разработчику целый зоопарк триггероподобных решений.

Однако в обработчиках типа Save или FlushDirty в качестве аргументов используются массивы состояний объекта. То есть изменять сам объект в них напрямую нельзя: в общем случае это просто не срабатывает, но могут быть и побочные эффекты. Нужно, ни много ни мало, поискать индекс элемента в массиве имён свойств объекта, затем по найденному номеру изменить значение в другом массиве текущего состояния объекта.

Рекомендуемая практика обхода ловушек такого рода – запрограммировать собственную защищённую (thread safe) очередь, куда складывать все созданные или изменённые объекты,а в событиях BeforeCommit или AfterCommit эту очередь обрабатывать.

Механизм прерываний также признан несовершенным, после чего был введён механизм событий (events), коим, начиная со второй версии, всем следует пользоваться.

В итоге имеем плохо документированную нестабильную систему, которая при отладке в разы труднее столь нелюбимых разработчиками триггеров баз данных. Тем не менее, если разработчик пишет код слоя домена, альтернатив практически нет. Генерация скелета кода по модели облегчает работу, но риски возникновения ловушек многопоточной обработки по-прежнему остаются на совести рядового программиста.

Использование обработчиков событий слоя домена должно быть рекомендовано еще в меньшей степени, чем использование триггеров слоя хранения данных.

Слой объектной абстракции доступа к реляционной СУБД в большинстве случаев скрывает не базу данных от приложения, а некомпетентность разработчиков приложения в области баз данных.

Чем меньше кода, тем лучше, это понятно. Например, качественная реализация слоя хранения использует DRI и прочую декларативность на уровне метаданных вместо императивного кодирования такой логики.

• Соотношение 1 к 1–2 примерно соответствует тому порогу, за которым начинается так называемый «плохой код».

• 1 к 3–4 – следует серьёзно заняться изучением вопроса переделки частей системы.

• 1 к 5 и более – надеемся, что случай нестандартный (сложные алгоритмы, распределенные вычисления, базовые подсистемы и компоненты реализуются самим разработчиком).

На мой взгляд, наиболее необычной и полезной была информация об ORM, триггерах, реляционных СУБД и скрещивании «ежа с ужом».

Широко известный в узких кругах своими несколько провокационными книгами67 публицист Николас Карр пишет: «Отделы ИТ в компаниях уходят в прошлое, а сотрудники соответствующих специальностей останутся не у дел из-за перехода их работодателей на сервис, предоставляемый по принципу коммунальных услуг.

Чем занимались тогда проектировщики, техники и организаторы? Примерно тем же самым, чем занимаются сейчас продвигающие на рынок системы «облачных» (cloud) вычислений, за исключением затрат на рекламу. Создавали промышленные вычислительные системы, отраслевые стандарты и их реализации.

Облако – оно же ВЦКП для корпоративного и даже массового рынка.

В чем состояли просчёты советской программы ВЦКП? Их два. Один крупный: проглядели стремительную миниатюризацию и, как следствие, появившееся обилие терминалов. Хотя отдельные центры, как, например, петербургский «ВЦКП Жилищного Хозяйства», основанный в 1980 году, действуют до сих пор, мигрировав в 1990х от мейнфреймов к сетям ПК. Второй: просчитались по мелочам, не спрогнозировав развал страны с последующим переходом к состоянию технологической зависимости.

Авторам мемуаров о создании в СССР первых ВЦКП [7], возможно, будет не только досадно, но и приятно. Досадно за опередившие время своего технологического воплощения концепции. Приятно за реализацию идеи – лучше поздно, чем никогда.

Программировать распределённое приложение сложнее, чем централизованное. Не только технически, но и организационно. Но в качестве выигрыша получаем автономию сотрудников, рабочего места и отсутствие необходимости поддержки службы в состоянии 24Ч7.

Наиболее очевидное применение децентрализации – автономные программы аналитической обработки данных. Мощность терминала позволяет хранить и обрабатывать локальную копию части

общей базы данных, используя собственные ресурсы и не заставляя центральный сервер накаляться от множества параллельных тяжёлых запросов к СУБД.

Услуга от продукта, материального товара, принципиально отличается минимум по трём пунктам:

•услуги физически неосязаемы;

•услуги не поддаются хранению;

• оказание и потребление услуг, как правило, совпадают по времени и месту.

Цепочка «производитель – конечный потребитель» подразумевает, что потребитель сам приобретает нужную программу и право на использование, сам её устанавливает и эксплуатирует. Разумеется, между производителем и потребителем может быть много перепродавцов и посредников, а вокруг – консультантов.

В схеме «программа как услуга» посредник (ВЦКП) берет на себя эксплуатацию программного продукта, то есть его установку, настройку, содержание, обновление и т. п., предлагая конечному пользователю услугу по доступу к собственно функциональности этой программы.

Предшественником современной СОА на базе веб-служб, с полным правом можно считать CORBA. Это сейчас задним числом обобщают подходы, представляя СОА как набор слабосвязанных сетевых служб, реализовать которые можно по-разному.

CORBA имела очевидные достоинства, прежде всего это интероперабельность77 и отраслевая стандартизация не только протоколов (шины), но и самих служб и общих механизмов – facilities.

Почему же и CORBA 2, продержавшись несколько лет в основном потоке технологий, была оттеснена на обочину? Хотелось бы выделить следующие:

•Стандарт поддерживался многими корпорациями, поэтому развитие требовало длительного согласования интересов всех участников. Некоторые из них продвигали свои альтернативы, например, Sun Java RMI и Jini, Microsoft DCOM.

•Несмотря на реальную поддержку интероперабельности, множества языков и сред программирования, основным средством разработки оставался C++.Но код на C++, манипулирующий инфраструктурой CORBA и службами, является излишне сложным по сравнению с той же Java или даже Delphi. Практиковавшие подтвердят, остальным будет достаточно взглянуть на примеры в Сети. А Javaпрограммисты не спешили использовать CORBA из-за упомянутых альтернатив.

•Запоздалая (1999 год), объёмная и весьма сложная спецификация компонентной модели. Javaсообщество к тому времени обладало альтернативой в виде EJB78 с открытыми и коммерческими реализациями.

Постепенно из кустов начали выкатывать на сцену рояль веб-сервисов, который из-за проблем с CORBA должен был стать новой платформой интеграции служб и приложений в гетерогенной среде. Концепции быстро придумали многообещающее название СОА. Поскольку это была технология, хотя и весьма базового уровня, но относительно простая, дешёвая и открытая не только на уровне спецификаций, но и в виде множества реализаций.

Если в частных случаях возникали вопросы нагрузки на поддержку соединений при большом количестве клиентов, они решались так же просто, как и в среде СУБД: приложение самостоятельно отсоединялось от сервера, выполнив пакет необходимых запросов. При желании нетрудно было также организовать и принудительное отсоединение по истечении заданного периода пассивности.

Для решения проблемы в рамках ООП необходим шаблон «Единица работы» (unit of work), суть которого в передаче веб-службе сразу всего упорядоченного множества объектов, подлежащих сохранению в управляемой сервером транзакции.

В общем случае шаблон является аналогом пакетной обработки транзакций, необходимой для сокращения времени жизни единичной транзакции.

Вторым «упрощением» стал переход от понятных прикладному программисту деклараций интерфейсов объектов и служб на языке IDL82 к WSDL83 – описаниям, ориентированным, прежде всего, на обработку компьютером.

Третьим «усовершенствованием» стал отказ от автоматической подгрузки связанных объектов84 в пользу исключительно ручного управления процессом.

На самом деле происходит обращение к серверу, вызов соответствующего метода у серверного объекта и возврат результата на клиента с возможным обновлением состояния локальных полей заглушки.

Сила CORBA проявляется в том, что технология может работать и как в приведённых примерах, то есть с реализацией элементов полноценного многопоточного сервера приложений, и аналогично веб-службам, обрабатывая в сервисах объявленные в интерфейсах структуры, напоминающие DTO.

Ожидает ли нас новое пришествие CORBA в виде облегчённой её версии – покажет время.

ЭВМ вымерли или вымирают

Две трети опрошенных отводят мейнфреймам стратегическую роль, которая только увеличивается

Наиболее ходовым термином в софтостроении является «новые технологии». По умолчанию новые технологии олицетворяют прогресс. Но всегда ли это так? Даже если не всегда, то в условиях квазимонополий, поделивших рынки корпораций, отказаться от «новых» технологий рядовым разработчикам непросто, особенно работающим в сфере обслуживания под руководством менеджеров среднего звена с далёким от технического образованием, оперирующих понятиями освоения и расширения бюджета и массовости рынка специалистов, а не технологической эффективностью.

Давайте подумаем, кто же выигрывает в этой гонке кроме самой NET, пользующейся своим квазимонопольным положением:

•Услуги по сертификации. Прямая выгода.

•Консультанты и преподаватели курсов. Сочетание прямой выгоды с некоторыми убытками за счёт переобучения и очередной сертификации.

•Программисты в целом. Состояние неопределённости. Выбор стоит между «изучать новые возможности» и «решать задачи заказчиков». А если изучать, то как не ошибиться с перспективой оказаться у разбитого корыта через пару лет.

•Разработчики в заказных проектах. Прямые убытки. За пару лет получен опыт работы с технологиями, признанными в новом фреймворке наследуемыми, то есть не подлежащими развитию, хорошо, если поддерживаемыми на уровне исправления критичных ошибок. А ведь всего 2–3 года назад поставщик убеждал, что эти технологии являются перспективными, важными, стратегическими и т. п. Необходимы новые инвестиции в обучение персонала и преодоление появившихся рисков.

• Разработчики продуктов. Косвенные убытки. В предлагаемом рынку продукте важна функциональность и последующая стоимость владения. На чём он написан – личное дело компании-разработчика. Тем не менее заброшенную поставщиком технологию придётся развивать за свой счёт

или мигрировать на новую. Скорее второе: в 2012 году по прежнему работает приложение 15летней давности, использующее DDE88, тогда как совместимость OLE Automation между версиями Office не гарантирована.

2–3 года – минимальный срок для появления первых промышленно работающих систем и, соответственно, специалистов по их разработке. Поэтому нормальный цикл концептуальных изменений 5–7, а то и 10 лет.

Широко распространено мнение о том, что Linux на порядок хуже, чем Windows поддерживает всякие периферийные устройства, но это не совсем так.

На мой взгляд, наиболее необычной и полезной была информация о COA и её предшественнике на базе веб-служб- CORBA, о NET и о том, что Linux на порядок лучше, чем Windows поддерживает всякие периферийные устройства. Честно говоря, с каждым днём все больше и больше хочу попробовать Linux.

Проектирование и процессы

Корпоративные информационные системы (КИС) прошли долгий путь от полной закрытости сплавленных с аппаратурой монолитов до создания модульных и открытых систем. Однако теперь вместо стандартизации процессов и реализации лучших практик создания базовой функциональности на передний план выходят конкурентные преимущества за счёт дифференциации и специализации.

Краткий словарь для начинающего проектировщика

Основная задача проектировщика – поиск простоты. Очень просто делать сложно, но очень сложно сделать просто. Начинающий проектировщик осознает это сам со временем, а пока нужно учиться элементарным понятиям для понимания.

Слоистость и уровни

Разбираться в слоях и уровнях должны не только разработчики КИС, то есть «скелета», но и те программисты.

Определение автоматизированной информационной системы (АИС) складывается из трёх основных её компонентов: людей, информации и компьютеров. Любая АИС – это люди, использующие информационную технологию средствами автоматизации [9]. И КИС не исключение.

Любая автоматизированная информационная система может быть рассмотрена с трёх точек зрения проектировщика:

- концептуальное устройство93;
- логическое устройство;
- физическое устройство.

Концептуальное устройство

Концептуальное устройство АИС составляют всего три слоя.

Слои концептуального устройства существуют в любой, подчёркиваю – в любой, информационной системе, даже если с точки зрения физической архитектуры или конкретного программиста их трудно различить.

Логическое устройство

Слои логической архитектуры не являются строго определёнными. Основным способом их выделения является постоянный диалог проектировщика с требованиями к системе.

Логическое устройство является предметом синтеза, на выходе стадии – технический проект. Логическое устройство АИС в разрезе концептуальных слоёв может выглядеть, как на рис. 5.

Физическое устройство

Аналогично логической архитектуре, физическое устройство тоже является предметом синтеза на стадии проектирования реализации.

Тонким клиентом (thin-client) традиционно называют приложение, реализующее исключительно логику отображения информации.

В противоположность тонкому, толстый клиент (rich-client) реализует прикладную логику обработки данных независимо от сервера.

Так называемый «умный» (smart-client) клиент по

сути остаётся промежуточным решением между тонким и толстым собратьями. Будучи потенциально готовым к работе в режиме отсоединения от сервера, он кэширует данные, берет на себя необходимую часть обработки и максимально использует возможности операционной среды для отображения информации.

У веб-браузера, как программируемого терминала, очень скромные, по сравнению с автономным приложением. Компромиссным решением является так называемое «насыщенное интернетприложение»94, также являющееся тонким клиентом, но обладающее всеми возможностями отображения клиента толстого.

Уровни

Даже в простой программе типа записной книжки имеется минимум 2 уровня:

- Уровень приложения, реализующий функционал предметной области.
- Уровень служб, поддерживающих общую для всех разрабатываемых приложений функциональность. Например, метаданные, безопасность, конфигурация, доступ к данным и т. д.

Совмещение

Реализация разных подсистем может осуществляться независимо друг от друга после спецификации своих межуровневых и межслойных интерфейсов.

Многозвенная архитектура

Концептуальных слоёв в автоматизированной информационной системе всегда три: хранения данных, их обработки и отображения. А вот физических, их реализующих, может быть от одного, в виде настольного приложения с индексированным файловым хранилищем, до теоретической бесконечности.

У каждой архитектуры есть свои преимущества и недостатки

Сегодня доля специалистов, имеющих опыт в системном проектировании, в общем потоке становится

всё меньше. Поэтому декомпозиция и выбор архитектуры часто проводится по принципу «прочитали статью, у этих парней получилось, сделаем и мы так же».

От автора: Я снова столкнулся с Novell в 2010 году в рамках небольшого проекта для французской национальной сети телевещания. Корпоративная система безопасности для тысяч компьютеров на

разных территориальных площадках была по-прежнему построена на службе каталогов NetWare, хотя и в тесной интеграции с аналогичной службой Microsoft. Новые компьютеры с Windows Vista/7 включались в общую систему. Сама Novell в конце того же 2010 года была куплена малоизвестной компанией Attachmate за целых 2,2 миллиарда долларов и формально прекратила существование. По некоторым сведениям, за Attachmate стояла Microsoft, незадолго до того выложившая 450 миллионов на приобретение у Novell технологий.

КИС, реализующая основные функции автоматизации деятельности торгово-производственной фирмы среднего размера: от бухгалтерии и складов до сборочного производства и сбыта – была разработана командой из 4–5 человек примерно за полтора года, включая миграцию с предыдущей версии. Система критичная, даже короткий простой оборачивается параличом деятельности фирмы.

Причина столь сжатых сроков? Ясное понимание решаемых прикладных задач, создание соответствующего задаче инструментария, прежде всего, языка бизнес-правил высокого уровня, и подтверждение тезиса Брукса о многократно превосходящей производительности хороших программистов по сравнению с остальными.

В другом случае четыре с половиной программиста сумели в короткие сроки создать и в течение многих лет сопровождать КИС для французского туроператора национального (один из крупнейших) и европейского уровня. Это уже потом к ней приделали веб-интерфейс для заказов клиентов и B2B10бшлюз с партнёрами.

Есть о чём призадуматься, особенно желающим начать новый проект

На мой взгляд, наиболее необычной и полезной была информация о АИС, КИС. Удивил тот факт, что КИС, реализующая основные функции автоматизации деятельности торгово-производственной фирмы среднего размера: от бухгалтерии и складов до сборочного производства и сбыта – была разработана командой из 4–5 человек примерно за полтора года.

Ultima-S – КИС из коробки

Существует два основных подхода к разработке КИС, условно называемых «от производства» и «от бухгалтерии».

В первом случае функциональным ядром системы становится планирование ресурсов производства.

Альтернативный путь проходит через бухгалтерию. Под термином «бухгалтерия» имеется в виду прежде всего внутренний, управленческий учёт на предприятии, а не фискальная её часть.

Поскольку основными потенциальными клиентами Ultima-S были именно оптово-розничные торговые компании, то функциональная архитектура системы базировалась на подходе «от бухгалтерии».

Каждый физический слой увеличивает трудоёмкость разработки, тестирования и последующих модификаций системы. Поэтому, с учётом предполагаемого тиражирования, развивать продукт было решено в следующих направлениях:

- минимизация числа звеньев;
- «утончение» клиентского приложения, в идеале до уровня веб-браузера;
- использование промышленной СУБД для реализации бизнес-логики;
- реализация некоторых механизмов ООП для упрощения разработки прикладными и сторонними разработчиками методом надстраивания новых классов.

Превратить промышленную СУБД в сервер приложений с технологической точки зрения просто. Для этого вам необходимо:

- запретить прямой доступ к таблицам базы данных;
- реализовать прикладную логику в виде хранимых процедур, функций и триггеров;
- разрешить доступ всех приложений только к соответствующим хранимым процедурам.

В технологии с тонким клиентом дополнительно потребуется:

- разработать протокол прикладного уровня для взаимодействия тонкого клиента и сервера приложений;
- запретить доступ ко всем объектам базы данных вообще, за исключением нескольких реализующих этот протокол хранимых процедур и, возможно, буферных таблиц.

Наконец, для надстройки над процедурным расширением SQL объектно-ориентированной среды, управляющей объектами предметной области, понадобилось:

- добавить поддержку декларации классов на уровне метаданных;
- реализовать механизм обработки сообщений между объектами;
- разработать набор базовых классов уровня ядра

и служб системы (см. уровни).

В общем случае, для создания программистом новых классов в системе было необходимо:

- декларировать классы;
- создать соответствующие таблицы;
- описать возможные ограничения на уровне метаданных (например, папку для создания по умолчанию или максимальное число ссылок);
- написать обработчики стандартных событий;
- добавить привилегии, видимые администратору;
- если необходимо, инициализировать данные, например, создать служебные объекты или документы-классификаторы.

Номенклатура базовых классов и системных служб позволяла пользоваться полуфабрикатами прикладного назначения: от управления жизненным циклом документа, доступом к папкам до бухгалтерских абстракций, позволявших задействовать механизмы материального и финансового учёта.

Следуйте совету Джобса: «Обычные художники заимствуют – великие воруют».

Учитесь у лидеров рынка, их успех не от «просто так».

На деле Ultima-S показала, что идея реализации сервера приложений средствами СУБД жизнеспособна и эффективна, также она совместима с лучшими функциональными практиками на рынке, как, например, моделирование в 1С. Но отсутствие маркетолога на проекте в качестве его лидера не дало

технологиям добиться коммерческого успеха, который пришёл уже к системам – наследникам Ultima-S.

С прикладной разработкой всё относительно просто: есть функционал, оцениваемый заказчиком или маркетологами в конкретную сумму, есть трудозатраты на его реализацию, остальное – прибыль от деятельности.

Ключевой особенностью платформы является возможность анализа хозяйственной деятельности предприятия практически в реальном времени. Как только документ оперативного контура меняет состояние «черновика» на одно из действительных, информация отражается на счетах управленческого учёта с заданными разрезами.

Нешаблонное мышление

Моя профессиональная практика складывалась таким образом, что обсуждение обобщённых решений касалось прежде всего задач предметной области и соответствующего уровня абстракций. С некоторой натяжкой их можно отнести к аналитическим шаблонам, так как в качестве основы брались только существенные части структур, интерфейсов или даже просто рабочие идеи. Лучше назвать их аналитическими эскизами.

О типовых решениях уровня реализации речь заходила редко, но уже в середине 1990х годов иногда возникали упоминания книги GoF – «банды четырёх»110[6] о приёмах объектно-ориентированного проектирования, где была предпринята попытка их обобщения.

Дело в том, что шаблоны из книги – уровня реализации и к моделированию предметной области имеют далёкое отношение. Они востребованы скорее теми, кто программирует (не хочу писать «кодирует») в группе, руководствуясь общей для всех подробной функциональной спецификацией. «

По моим представлениям, книга имела к науке весьма опосредованное отношение и являлась скорее чем-то вроде поваренной книги.

Сложнее обстоит дело с теми, кто только начинает свой путь. Прочтение сего труда новичком, как мне кажется, является прямым аналогом попадания в прокрустово ложе диктуемой парадигмы. Потому что книга описывает набор решений, а неокрепшему за недостатком практики уму проектировщика надо научиться самому находить такие решения и пути к ним.

Оказывается, шаблоны сыграли с программистским сообществом злую шутку: они оказались ещё и зависимыми от языка программирования.

Преобразование типов делает его сложным для сопровождения, кроме того, нет никакой гарантии, что проверка для базового класса не выполнится раньше, чем для производного. Достаточно ошибиться в порядке следования операторов if, и на ветку производных классов программа никогда не попадёт.

Если разные классы элементов документов имеют полиморфные свойства, собираемые статистикой, то задача ещё более усложняется.

Итого на простом примере имеем целый букет решений вместо одного шаблонного, из которых можно выбирать оптимальный.

«Thinking in patterns». В переводе на русский язык –

«Мыслить шаблонно». Ещё недавно привычка шаблонно мыслить считалась в инженерном сообществе признанием ограниченности специалиста, наиболее пригодного для решения типовых задач с 9 утра до 6 вечера. Теперь не стесняются писать целые книжки о том, как научиться шаблонному мышлению…

Думать головой

Откажитесь от термина «наследование», который искажает смысл действий. Мы обобщаем. Обобщение (наследование) реализации или интерфейсов – весьма неоднозначный механизм в объектно-ориентированном программировании, его применение требует осторожности и обоснования.

ОСНОВНОЕ ПРАВИЛО

Обобщаемые классы должны иметь сходную основную функциональность.

Например, если два или более класса:

- порождают объекты, реализующие близкие интерфейсы;
- занимаются различающейся обработкой одних и тех же входных данных;
- предоставляют единый интерфейс доступа к другим данным, операциям или к сервису.

Почему шаблоны «вдруг» стали ненужными? Потому что требуется только обобщение, причём в перечисленных случаях необходимость операции более чем очевидна. Требования же исходят напрямую из вашей задачи. Корректно проведя обобщение, вы автоматически получите код и структуру, близкую к той, что вам предлагают зазубрить и воспроизводить авторы разнообразных учебников шаблонов.

Ошибка приводит к построению иерархии по неосновному признаку. Когда внезапно найдётся ещё один такой признак, возможно, более существенный с точки зрения прикладной задачи, обобщить больше не удастся: придётся ломать иерархию или делать заплату в виде множественного наследования, недоступного во многих объектно-ориентированных языках. Или пользоваться агрегацией.

ВОЗЬМИТЕ ЗА ЭМПИРИЧЕСКОЕ ПРАВИЛО

Глубина более двух уровней при моделировании объектов предметной области, вероятнее всего, свидетельствует об ошибках проектирования.

Про сборку мусора и агрегацию

Наличие в некоторых языках механизма сборки мусора, является примером отказа от самой идеи справиться с этими

проблемами и молчаливым признанием возможности присутствия в среде объектов, не принадлежащих ни подпрограммам, ни другим объектам.

Итак, сборщик мусора, он же GC – garbage collector в средах программирования с автоматическим управлением памятью. Наиболее очевидное преимущество – программисту не надо заботиться об освобождении памяти. Хотя при этом все равно нужно думать об освобождении других ресурсов, но сборщик опускает планку требуемой квалификации и тем самым повышает массовость использования среды. Но за все приходится платить. С практической стороны недостатки сборщика известны, на эту тему сломано много копий и написано статей, поэтому останавливаться на них я не буду. В ряде случаев недостатки являются преимуществами, в других – наоборот. Черно-белых оценок здесь нет. В конце концов, выбор может лежать и в области психологии: например, кто-то не любит, когда компьютер пытается управлять, не оставляя разработчику достаточных средств влияния на ход процесса.

Наиболее распространённой ошибкой является сохранение

ссылок на эти объекты в другом объекте вне контейнера. При этом часто оказывается, что ссылки ещё живы, но указывают в пустоту, потому что контейнер уже удалён.

Решение здесь достаточно простое.

НУЖНО ВЗЯТЬ ЗА ПРАВИЛО, ЧТО

контейнер всегда управляет своими объектами. Поэтому обращаться к его внутренним объектам нужно только через интерфейс самого контейнера.

сборка мусора, позволяющая программистам избавиться от назойливых ошибок обращения к памяти (access violation), является лишь фиговым листочком, прикрывающим до некоторого момента всё те же старые проблемы.

На мой взгляд, наиболее необычной и полезной была информация о Ultima-S, «шаблонном мышлении». Так же полезны и правила, приведенные в данной части книги. Произвела впечатления критика в отношении “сборщика мусора”.

Журнал хозяйственных операций

Вопросы, подобные «почему нельзя хранить остатки в форме текущих величин», «зачем нужна история операций», «зачем там нужна транзакция в режиме «сериализация» (что такое режим serialized можно прочитать в статье [20]), не раз всплывали в дискуссиях, поэтому я кратко расскажу об этом в рамках отдельной главы, чтобы в следующий раз просто ссылаться.

Анти-шаблон “Таблица остатков”

Приходит клиент и говорит: «Мне выписали 10 штук, а на складе только 8, я на вас, жуликов, в суд подам». Парадокс? Никакого парадокса. За товаром он пришёл сегодня, но продали ему товар вчера. А на состояние «вчера» после корректировки остаток был бы отрицательным. Вот ему и не хватило. Даже для фактических операций нужно считать остаток по истории (журналу). Не говоря уже о резервировании товара, где ситуация меняется гораздо быстрее: то тут отменили, то там подтвердили. Считать по журналу может быть долго; необходимо защитить считанные значения, чтобы при последующей записи не возникло «минусов».

сути, это и есть та самая сериализованная транзакция. Она гарантирует, что считанные значения не будут изменены другой транзакцией. То есть продажа не будет давать отрицательный остаток, если между операцией расчёта остатка и расхода вклинится другой. Если же ваши операции расчёта, проверки и расхода работают в одной транзакции на уровне изоляции «сериализация», то такая ситуация исключена.

Почему нельзя просто блокировать всю таблицу-журнал, а надо использовать какие-то хитроумные транзакции с непонятным уровнем изоляции?

Если заблокировать таблицу, то второй продавец всё равно будет ждать первого. Всегда. Использование транзакции в режиме serialized вполне может привести на физическом уровне к блокировке всей таблицы.

«Зачем хранить историю остатков?».

Это, мягко говоря, нехорошо. К тому же, если ваш расчёт не написан на языке СУБД – SQL.

Совет начинающим: учите сиквел, транзакции, уровни изоляции, и будет ваша система быстрой и надёжной. И не только в примере с учётной системой, но и вообще по жизни. Опытным же разработчикам есть поле для оптимизации и дальнейшего совершенствования решений, включая альтернативные подходы.

UML и птолемеевские системы

В UML должно настораживать уже самое первое слово – «унифицированный». Не универсальный, то есть пригодный в большинстве случаев, а именно унифицированный, объединённый. Силами небольшого количества экспертов, если точнее, сначала двух, позже к ним присоединился третий, был создан сборник, содержащий наиболее удачные нотации из их собственной практики. Сборник стали продвигать в массы, а за неимением лучшего, и в стандарты.

Модель от картинки в графическом редакторе отличается наличием базового формализма, позволяющего проверить созданную конструкцию на непротиворечивость, а если повезёт, то и на полноту.

UML до языка ещё далеко т.к. язык можно автоматически проверить на формальную правильность. Входящий в UML OCL хоть и является языком, но выполняет лишь малую часть работы по проверке, и не собственно модели, а её элементов.XML тоже не язык, а технология создания языков пользователя на базе разметки и формальных правил её проверки посредством DTD или схем.

Итак, 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.

Поэтому использование UML имеет две основные альтернативы, напрямую зависящие от целей:

•Цель – использовать «как есть». Не заниматься вопросами целостности, ограничившись рисованием частей системы в разных ракурсах. Если повезёт, то часть кода можно будет генерировать из схем.

•Цель – использовать для моделирования и генерации кода. Придётся создать свои формализмы, соответствующие моделируемой области. В самом минимальном варианте – использовать в принудительном порядке стереотипы и написанные руками скрипты и ограничения для проверки непротиворечивости такого использования.

Формализация модели является, по сути, созданием предметно-ориентированного языка.

С картинками в UML сложилась некоторая неразбериха вкупе с излишествами. Например, диаграммы деятельности (activity diagram), состояний (state machine diagram) и последовательностей (sequence diagram), а также их многочисленные подвиды по большому счёту описывают одно и то же – поток управления в программе.

В UML рекомендуется использовать комментарии для отражения требований к системе, а сами прецеденты неформальны. При отсутствии проработанной системы стереотипов приходится, например, отделять приходные документы от расходных с помощью раскраски.

Анализом прецедентов в UML. Это идеальный инструмент для создания птолемеевых систем. Только если модель Птолемея является геоцентрической, то прецеденты использования – модель заказчико-центрическая.

Прецеденты, как и геоцентрическая модель, могут удовлетворительно описывать явление, не касаясь его сущности. Всякий раз разработчики очередного корпоративного приложения создают свою «птолемеевскую систему», которая при попытке примерить ее на другую фигуру внезапно оказывается неподходящей и нуждается в серьёзной перекройке.

В RUP рекомендуют не увлекаться наворачиванием прецедентов друг на друга (avoiding functional design).

Из положительных моментов UML и поддерживающих его сред необходимо отметить, что в ситуации, когда задача складывается, как мозаика, из обрывочных сведений представителей заказчика, прецеденты помогут хоть как-то формализовать и зафиксировать неиссякаемый поток противоречивых требований. При отсутствии экспертизы в предметной области риски недопонимания и недооценки снижаются. Диаграмма классов UML – при определённых усилиях и дисциплине программистов поможет вам автоматизировать генерацию части рутинного кода приложения.

«Двусторонние» же инструменты (twoway tools), тесно интегрированные со средами программирования и позволяющие непосредственно во время разработки оперировать диаграммами с одновременным отражением изменений в коде и, наоборот, импортировать меняющийся руками код в модель. Таковы, например, ModelMaker или Together. Но если для повышения продуктивности отдельного программиста такой подход даёт хорошие результаты, то необходимость синхронизации моделей при коллективной работе над общим кодом может свести преимущества к минимуму.

Для тиражируемых продуктов и ориентированного на специализации проектного софтостроения наиболее интересен подход с разработкой собственных языков, где необходимость использования UML неочевидна.

На мой взгляд, наиболее необычной и полезной была информация об UML, а так же какие-то ответы на “наболевшие” вопросы и,конечно, полезные советы начинающим.

Теперь, спустя немногим более десяти лет, когда начинают всплывать проблемы с поддержкой из-за отсутствия специалистов, принимаются решения о переделке систем в новой технологии с минимальными изменениями функциональной полезности или вовсе без оной.

Побитовые операции с типами данных разной длины не являются сильным местом языка и платформы в целом.

Первая проблема была решена односторонней синхронизацией локальных данных с соответствующей их частью в хранилище.

Сначала на SQL, но с курсорами, получив ещё худший результат. Потом попытались избавиться от курсоров, но всё равно не смогли достигнуть требуемой производительности.

Физическая архитектура хранения, секционирование таблиц, оптимизация индексов и запросов по выборочной статистике – эти вопросы всерьёз не рассматривались, база данных тихо кряхтела одним файлом, правда, на мощном 16ядерном сервере с 32 гигабайтами памяти и быстрым дисковым массивом.

При правильной организации физического хранения и грамотном SQL-коде как минимум тех же результатов можно было бы добиться, не выходя за пределы реляционной БД.

Очередной урок, «кейс», экспериментаторам с единственно правильными архитектурами, любителям городить новые слои, чтобы спрятать за ними свою некомпетентность в области СУБД.

На мой взгляд, наиболее необычной и полезной была информация о распределённом анализе данных.

Code revision, или Коза кричала

Такой универсализм стал причиной использования мега-справочника одновременно для хранения внутренних счётчиков нумерации записей: текущая величина хранилась в строковом поле колонки «Значение» в формате «префикс; текущий номер

Кроме перечисленных манипуляций со строкой, вначале делается попытка заблокировать запись через соответствующую опцию SQLзапроса.

В коде формы есть только одно место, где значения кнопок используются.

Дж. Фокс [2] выводит из своего опыта проектной работы в IBM важную мысль, что большой ошибкой является привлечение к процессу внутреннего тестирования и обеспечения качества посредственных программистов. По его мнению, компетентность специалиста в этом процессе должна быть не ниже архитектора соответствующей подсистемы.

Качество кода во многом зависит от степени повторного использования.

гибкой (agile) разработки программ достаточно рискованно.

В достаточно консервативных банковских или промышленных приложениях «водопад» может подойти и для комплексной системы.

Ключевой особенностью спиральной технологии является прототипирование. В конце каждого витка после этапа стабилизации заказчик получает в своё распоряжение ограниченно работающий прототип целой системы, а не отдельных функций. Основная цель прототипа состоит в максимально возможном сближении взглядов заказчика и подрядчика на систему в целом и выявлении противоречивых требований.

Спиральная модель не навязывает присутствие всех стадий на каждом витке.

Слабым звеном в спиральной методологии является определение длительности очередного витка, его стадий и соответствующее выработанному плану управление ресурсами

Ключевой особенностью гибкой методики является наличие мифологического титана – владельца продукта (product owner), который лучше всех знает, что должно получиться в итоге.

В успешно выполненный в «водопадной» схеме проект может быть также

В большинстве случаев выполнен в «гибкой» разработке.

Современная тенденция – огромное число некритичных проектов, программ и систем-пристроек к основной КИС. Масштаб проектов небольшой (сотни тысяч строк кода), заказчик точно не знает, что хочет получить в итоге, а подрядчик не имеет опыта в данной предметной области, если вообще имеет хоть в какой-то, и поэтому не может ему объяснить, что кактусы за полярным кругом не растут

Сложилось впечатление, что модульные тесты хорошо использовать для продуктов, которые реализуют какой-то стандарт или спецификацию (СУБД, веб-сервер), но для тиражируемого веб-приложения - это смертельно. Причины:

1. При изменении спецификаций затраты времени на приведение в актуальное состояния тестов могут быть куда больше, чем на собственно код.

2. Сбои, которые могут выловить тесты

6. Я совсем не уверен, что пользователи хотят получать новую версию раз в 2 недели.

7. Для корпоративного софта пользователи хотят длительного периода поддержки, пара лет, минимум.

Излишне религиозная атмосфера превратила вполне здравую и работающую с 1970х годов технологию модульных тестов в настоящий каргокульт.

Разработка модульных тестов – это тоже разработка.

В отличие от тестов функциональных, завязанных на интерфейсы подсистем, модульные тесты требуют переработки одновременно с рефакторингом рабочего кода. Это увеличивает время на внесение

изменений и ограничивает их масштаб, приучая разработчика минимизировать реструктуризацию, заменяя её надстройкой, быстро трансформирующей архитектуру в Ад Паттернов.

Модульные тесты тоже бывают сложными, а значит, с высокой вероятностью могут содержать ошибки.

Модульный тест – это самый нижний уровень проверок.

Качество программного продукта – многозначное и сложное понятие. Производственная культура – ещё более сложное. В одном можно быть уверенным: ни о какой культуре софтостроения не может идти и речи, если любой программист из коллектива не способен остановить бессмысленный циклический процесс для выяснения, какого же рожна по историям заказчика потребовалось обобщать четырёхногих коров и обеденные столы.

На мой взгляд, наиболее необычной и полезной была информация о методологиях, модульном и гибкой (agile) разработки.

Приключения с TFS

TFS 2008, как выяснилось, требует для себя:

- SQL Server 2005 или выше;
- Reporting Services 2005 или выше;
- Analysis Services 2005 или выше;
- Sharepoint Services не ниже 3 SP1;
- Internet Information Server 6 или выше; •.NET 3.5 SP1;
- ещё - по мелочи вроде конкретных хот-фиксов.

По мере изучения сего списка я постепенно впадал

в прострацию, сравнивая с ничего не требующим пакетом SVN.

Гетерогенная сеть хоть и интегрируется с Active Directory, но не до конца: интегрированная безопасность (integrated security) для клиентов не работает, так как они входят в сеть под профилем NetWare. Ну, и ладно бы с этим, ведь создать регистрации на небольшую группу в TFS несложно, как и в SVN. Плохо другое, доступ с компьютера либо в Интернет с аутентификацией через прокси, либо в локальную сеть к серверам. То есть работать в режиме переключения

с терминала на Интернет в браузере с обычного рабочего места нельзя, эта опция доступна только с компьютеров администраторов в отделе поддержки.

Только в официальной документации об этом не написано, нужен консультант, желательно от самого Microsoft. Вызов консультанта с предварительным

утверждением бюджета – это ещё неделя простоя.

Принято решение ставить все на один виртуальный сервер.

Переустанавливаем на него: SQL Server, Reporting Service, Analysis Services. СУБД и все компоненты на строены и работают.

Главное – репозиторий исходников работает. Почти как на SVN.

Программная фабрика: дайте мне модель, и я сдвину Землю

Идея разрабатывать программы, минимизируя стадию кодирования на конкретных языках под заданные платформы, появилась достаточно давно. Прежде всего в связи с неудовлетворительной возможностью языков высокого уровня третьего поколения (3GL) описывать решаемые прикладные задачи в соответствующих терминах. За последнее время к этой причине добавилась ещё и поддержка независимости от целых платформ, ведь прогресс, как мы знаем, неотвратим, особенно «прогресс».

У программистов «от сохи» отношение к CASE, как правило, негативное на уровне «я не верю, что какие-то картинки генерируют код лучше написанного руками».

У более продвинутых программистов, имевших опыт написания и сопровождения тысяч строк однотипного рутинного кода, претензии становятся обоснованными и касаются, как правило, следующих сторон применения CASEсредств:

• Если ручное написание кода принять за максимальную гибкость, то CASE может навязывать каркас, стиль кодирования и шаблоны генерации частей программ, ограничивающие не столько полёт фантазии программиста, сколько возможность тонкой настройки генерируемого кода.

• CASE работает только в условиях дисциплины, когда ручные изменения генерируемого кода исключены или автоматизированы (постобработка).

В качестве решения перечисленных проблем появились так называемые двусторонние CASEинструменты (two way tools), позволяющие редактировать как модель, непосредственно видя изменения в коде, так и, наоборот, менять код с полу– или полностью автоматической синхронизацией модели.

Следующим шагом в развитии автоматизированных средств софтостроения явилась программная фабрика – синтез подходов управляемой моделями разработки и архитектуры, генерирующий не только отдельные компоненты системы, но целые слои в соответствии с выбранной архитектурой и платформами.

В описании конфигурации джиннов видно, что его основу составляет сборка, один из классов которой, реализующий интерфейс IGenie, является точкой входа. Каждый джинн имеет как общие для всех параметры, например, каталог для выходных файлов, так

и специфичные, передаваемые через тег Param, описываемые в документации.

шаблоны реализации типовых задач уровня ядра и системных служб:

• Например, шаблон «Реестр объектов» добавляет к системе возможность ведения централизованного реестра всех создаваемых объектов.

• Шаблон «Версия состояния» является встроенной в NHibernate возможностью отслеживания конфликтов в многопользовательской среде.

• Шаблон «Аудит» в простейшем варианте является регистрацией для каждого хранимого объекта информации о времени его создания, последнем редактировании и авторе.

• Шаблон «Локализация» добавляет в генерируемый код возможность перевода сообщений в рамках технологии GNU gettext.

• Наконец, шаблон «Безопасность» в простейшем варианте ограничивает доступ к веб-службам через механизм аутентификации, логику которой необходимо реализовать в переопределяемом методе соответствующего класса.

Все классы домена являются расширяемыми (partial), что позволяет разработчику вынести специфичные не поддерживаемые моделью реализации свойств и методов классов в отдельные файлы.

Генерируемые для слоя веб-служб C#файлы предназначены для создания двух сборок: собственно, служб и интерфейсов к ним, используемых клиентами.

На мой взгляд, наиболее необычной и полезной была информация о TFS,шаблонах и из классификации, развития TFS.

Cherchez le bug, или Программирование по-французски

Хаос наступает внезапно

«баг» (от англ. bug – жучок) – это «жучок-вредитель» в программе, то есть ошибка, аномалия, сбой.

Система, с которой я начал работать, уже успешно прошла свой порог несколько месяцев назад и жила полнокровной, отдельной от авторов жизнью. Определить, что критический порог пройден, несложно, для этого у меня есть один простой признак: «Ты изменил чуть-чуть программу в одном месте, но вдруг появилась ошибка в другом, причём даже автор этого самого места

не может сразу понять, в чем же дело»128. Существует и второй признак, не менее практичный: «Смотришь на чужой текст программы, и тебе не вполне понятен смысл одной его половины, а вторую половину тебе хочется тут же полностью переписать».

Второй способ: 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. Третий способ – выдать желаемое за действительное, побольше маркетинга, шуму, «подцепить» несколько заказчиков и на их деньги попытаться всё-таки перейти ко второму или первому способу. 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».

Кстати, если программист говорит вам, что в данном месте программа «должна работать», это значит, что с очень большой вероятностью она не работает, а он её просто не проверял в этом месте.

Центральным звеном системы является модуль с типичным названием «ядро» (kernel). Многострадальное ядро, несмотря на месяцы групповых измывательств со стороны коллектива программистов с меняющимся составом, каким-то чудом все-таки компилировалось и запускалось

месяцев десять назад ядро состояло из кучки «ядрышек», которые просто слили в один кусок, когда на первых запусках время ожидания ответа не устроило

даже самих авторов. Однако после результатов такого слияния бдительность была потеряна надолго, хотя ничто не мешало параллельно с разработкой делать хотя бы простые тесты и прогоны.

Хорошо там, где нас нет

Зато вы можете быть спокойными: без работы в ближайшие десятилетия не останетесь.

О технических книгах

Дефрагментация мозгов

С. Дмитренко в предисловии к своему переводу известной книги Leo Brodie «Thinking FORTH. A Language and Philosophy for Solving Problems» в 1993 году писал:

Можно сказать, много грустных слов о тенденциозности современной около-технической

литературы, о переориентации отечественных программистов и разработчиков с исследовательских и новаторских на чисто коммерческие работы, о складывающемся монополизме отнюдь не лучших (зато более хватких) производителей компьютеров и программного обеспечения и т. д. И все же, несмотря на эти признаки нарастающего вырождения в нашем, да и мировом компьютерном деле, я надеюсь, что подъем наступит, и мы будем его свидетелями и реализаторами.

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

Но для возрастания значения книг на фоне всеобщей фрагментации знаний необходимо умение аудитории читать и усваивать длинные тексты, неуклонно снижающееся последние десятилетия.

Почему же относительно легко студентам? Дело, прежде всего, в смене образа мыслей, а может быть, и восприятия самой действительности. Мир – это такая очень большая и сложная компьютерная игра, а преподаватели, соответственно, должны обучать не премудростям стратегий познания мира, а практическим приёмам, секретным кодам и даже шулерству, чтобы пройти в этой игре на следующий уровень.

Изредка мне приходится практиковать обучение СУБД, где я столкнулся ровно с тем же явлением. Группа стажеров примерно моего возраста и старше всю неделю честно старалась вникнуть в детали, написанные мелким шрифтом после каждого слайда, уместить знания в некую систему, желательно, не диссонирующую с уже имеющейся в голове.

И напротив, аудитория примерно 20–25 лет оказалась совершенно равнодушна к пояснительному тексту. Если картинка слайда была удачной, то она более-менее откладывалась в памяти. Вдобавок, шло конспектирование некоторых случаев из моей практики с короткими кусками кода. Иное дело – лабораторные работы, когда решение находилось увлекательным методом «научного тыка». Если же метод не срабатывал, то ко мне обращались с просьбой подсказать «код доступа для прохождения на следующий уровень».

Является ли фрагментарное мышление приспосабливанием к многократно увеличившемуся потоку информации? Отчасти да, только это скорее не адаптация, а инстинктивная защита. Чтобы осознанно фильтровать информацию о некоторой системе с минимальными рисками пропустить важные сведения, нужно иметь чётко сформированные представления о ней, её структуре и принципах функционирования.

Итак, настоящий исследователь должен:

- быть достаточно ленивым. Чтобы не делать лишнего, не ковыряться в мелочах;
- поменьше читать. Те, кто много читает, отвыкают самостоятельно мыслить;
- быть непоследовательным, чтобы, не упуская цели, интересоваться и замечать побочные эффекты.

Технические книжки – это не бессмертные произведения графа Толстого. К ним должен быть суровый, сугубо индивидуальный и безжалостный подход без какой-либо оглядки на чужое мнение.

Теперь, по мере чтения, начинайте делать пометки

Если, пролистав с полсотни страниц, вы не сделаете ни одной пометки, то можете смело закрывать сие произведение. В топку бросать, наверное, не надо, но поберегите своё время. Если же к концу прочтения вам хватит листа А4, исписанного убористым почерком, для всех пометок, то можете считать, что чтение прошло не зря. Ну а если одного листа не хватило… Тогда смело идите на свой любимый интернет-форум и там поделитесь с коллегами своими находками

Разделение труда на «автора» и «кодировщика» в подобном процессе также присутствует. Даже если автор номинальный, издающий под своим брендом труд неизвестных литераторов, что, к сожалению, случается.

В итоге обнаруживаем немало сходства в процессе.

На мой взгляд, наиболее необычной и полезной была информация о читателе нашего времени, о его усваивании прочитанного; современной технической литературе; о разнице поколений; без работы в ближайшие десятилетия не останетесь, но для этого нужно активно развиваться самому, читать литературу, а для эффективного усваивания информации поможет ряд правил, описанных в последних главах этой книги.

Довольно-таки познавательная книга, хотя и со специфическими моментами. Если же оценивать в целом данную книгу, то информация довольно таки полезная и в некотором смысле даже жизненно необходимая для начинающего разработчика. Было описание маленькой фирмы, у которой были такие же проблемы, что и у ”больших”. Что мне лично нравиться в этом деле так это неограниченная возможность, своего рода свобода. Практически любая маленькая компания может “стрельнуть” главное- идея и инвестор. Закончить хочется фразой одной из глав этой книги: “Зато вы можете быть спокойными: без работы в ближайшие десятилетия не останетесь…”.

Лебедев Николай

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.

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.

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 Chrome 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 кроссплатформенных 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.

Часть 4.

- 1970 -1980 «C++ против Паскаля»
  В следствие (в основном C++), 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 C++ достаточно «тонкий» и человеческий фактор – решающий на проекте, появились более безопасные языки, похожие на C++: сначала Java, потом C#.

  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 с 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 с NHibernate автор вообще промахнулся. Было сказано, что работать с Nhibernate трудно и неудобно, так как 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. Используя 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 находиться 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.

Часть 5.

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...

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 C++ работать в этом плане было приятнее.

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.
2) 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.
3) Хранить историю остатков нужно.
   Чтобы найти значение остатка необходимо обработать все произведенные вычисления неслабым функционалом. При этом скорость вашей программы уменьшается в разы.
4) 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

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.

Часть 10.

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.

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 с 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.

Team Foundation Server (сокр. TFS) — продукт корпорации Microsoft, представляющий собой комплексное решение, объединяющее в себе систему управления версиями, сбор данных, построение отчётов, отслеживание статусов и изменений по проекту и предназначенное для совместной работы над проектами по разработке программного обеспечения. Данный продукт доступен как в виде отдельного приложения, так и в виде серверной платформы для 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)

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.

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.

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 с 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.

Михальцова Анна

О нашей профессии

Усовершенствование технологий. Неизменна лишь суть профессии программиста (человек, способный заставить компьютер решать поставленное перед ним множество задач). Расширение области применения компьютеров.

На смену ЭВМ очень быстро пришли компьютеры, которые позволяют выполнять практически любую задачу.

Специализация

Формулы, отражающие суть работы программиста. Программист как инженер, способный как самостоятельно формализовать задачу, так и воспользоваться стандартными средствами её решения на ЭВМ.

Роль математических дисциплин в работе программиста: "Исторически сложилось так, что многие программисты были преимущественно математиками и самостоятельно занимались формализацией задач."

Большая роль отводится инженеру. Всё-таки это не совсем программист. Возможно пару лет назад, в эпоху ЭВМ, так оно и было. Но сейчас инженер, в моем понимании, человек, имеющий больше отношения к конструированию, проектированию каких-то деталей, оборудования.

Кто такой ведущий инженер, или Как это было

Иерархия подразделений. Иерархия должностей. Уровни принятия проектных решений. Функциональные специализации (роли).

Различия между инженерами и техниками:

- Формальные (техников готовили в ПТУ и техникумах, их образование - среднее специальное; инженеров - в технических вузах. Были математики-программисты, которых готовили в университетах.);
- Фактические (техники не занимались постановкой задач и проектированием программных систем, ограничиваясь непосредственно программированием и эксплуатацией.).

Европейский взгляд на инженеров с инженерных школ и университетов.

Программирование: искусство, ремесло или наука? Иван Ефремов о красоте.

- "Нельзя «сделать красиво», если относиться к работе исключительно утилитарно и шаблонно. Но и нельзя «сделать красиво», если рассматривать софтостроение лишь как искусство и средство самовыражения."

6 миллионов на раздел пирога

"… Так что определённый прогресс все-таки налицо."

Переквалификация как способ сохранить рабочее место при сокращениях.

Гауссово распределение уровня профессиональной компетентности программистов. Проблема, связанная с соотношением зарплата-количество программистов. Самосовершенствование (в программировании) и творческий подход – залог успеха.

"Чтобы не просто зарабатывать на хлеб, но и мазать его маслом, сохраняя при этом возможности технического творчества, вам лучше держаться подальше от тех направлений деятельности, где конкурентами будут 6 миллионов человек.

Профориентация

Словарик ключевых фраз: быстро растущая компания, гибкие (agile) методики, умение работать в команде, умение разбираться в чужом коде, гибкий график работы, опыт работы с заказчиком, отличное знание XYZ.

"Лучше давать испытуемому некоторый нестандартный тест, чтобы просто посмотреть на ход его мысли."

Про мотивацию

Традиционные способы управления мотивацией: педагогика и воспитание. Отличия мотивации от стимуляции.

Технологии

"Доступность информации снизила ее значимость, ценность стали представлять не сами сведения из статей энциклопедии, а владение технологиями." И ведь действительно, в наше время, когда интернет является практически единственным источником получения информации, вопрос заключается только в том, чтобы найти какое-либо устройство (мобильный телефон, ноутбук, планшет и т.п.) и открыть браузер. Какая бы полезная информации не содержалась в книгах, она со временем устаревает (если это конечно не касается каких-либо физических законов, табличных неизменных значений и т.п.). Государственная власть разделяется на три ветви: законодательную, исполнительную и судебную. И пора уже официально признать четвертой властью СМИ.

Безысходное программирование

Любая программа базируется на исходном коде какого-либо языка программирования. Безысходное программирование подразумевает под собой программирование без исходного кода, то есть когда пишется код, но тот, кто его пишет, даже не представляет, как он функционирует на более глубоком уровне.

Без исходного кода возникают трудности в понимании внутренней логике работы программы. И тестирование такого кода во многих случаях просто бессмысленно, потому что нельзя с уверенностью предугадать реакцию системы.

Эволюция аппаратуры и скорость разработки

Не соглашусь с тем, что ООП вытеснило женский труд из компьютерной отрасли. Скорее уж это связано с неумением разбираться в технологиях софтостроения.

О карманных монстрах

"Поэтому новичкам не раз предстоит столкнуться с заданием типа "быстро добавить поле в форму" и познакомиться с внутренним устройством подобных программ – карманных монстров, готовых откусить палец неосторожно сунутой руки."

Ну так уж прям только новичкам?! А сколько раз и достаточно опытным программистам приходится "копаться" в абсолютно новом для них материале. Как будто "карманные монстры" страшны только начинающим. Чем больше материала усвоено, тем сложнее он остался. Поэтому и влияние ошибок наиболее весомо не на начальной стадии умения программировать.

ASP.NET и браузеры

"Всякий раз, когда приходилось что-то делать при помощи технологии ASP. NET или просто править чей-то код, даже правильно написанный, меня не покидало ощущение копания по локоть в большой столовской кастрюле с макаронами." Это верно и не только для технологии ASP.NET. Взять даже просто любой кусок кода для примера. Грубо говоря, это не твой стиль, не твои мысли. И счастье будет, если написано еще будет более или менее понятно, а не как на Ассемблере с его go to, хотя и в нем куда больше логики, чем в некоторых "кастрюлях с макаронами".

Апплеты, Flash и Silverlight

В этой главе много внимания было уделено Flash-приложениям. Всё-таки мне кажется они немного устаревают. В как плане? На сегодняшний день практически то же самое можно сделать с использованием HTML, JavaScript. Пусть это и будет немного тяжелее (особенно в количестве строк кода). Несомненно, преимущества есть у каждой из перечисленных технологий, просто мне ближе не Flash.

ООП – неизменно стабильный результат

"ООП позволяет увеличить количество кода…"

Увеличить ли? Как мне кажется, ООП наоборот помогает уменьшить его. За счёт классов программа выглядит гораздо структурированной.

"Язык С по-прежнему занимает первое место согласно статистике активности сообществ программистов, стабильно опережая второго лидера Java."

Не знаю, какая статистика имеется в виду, но судя по статьям в интернете и нашим обсуждениям на спецкурсах среди данных языков явный лидер Java.

ORM, или объектно-реляционный проектор

Есть масса альтернатив, но почему-то для хранения данных используются в основном базы данных. Интересно было бы поразмышлять, почему именно им отдают такое предпочтение. Ведь иногда (если программа пишется на Java, реализуя ООП как парадигму программирования) удобно хранить данных непосредственно в полях класса.

Как обычно используют ORM

"Аббревиатура SQL вызывает негативные ассоциации, связанные с чем-то древним..."

Да, возможно он уже устаревает. Но я бы всё-таки не стала так негативно отзываться, так как множество программистов всё ещё продолжают использовать именно SQL, а не хваленый автором книги LINQ.

Триггер как идеальная концепция для NHibernate

"Но надо отдать должное: ребята честно признались, что после моего ухода никто не сможет этот сиквел-код поддерживать и модифицировать. Вот, собственно, и главная причина первоначального выбора. Красноречивое подтверждение тезиса о том, что слой объектной абстракции доступа к реляционной СУБД в большинстве случаев скрывает не базу данных от приложения, а некомпетентность разработчиков приложения в области баз данных."
Очень верно подмечено. Когда-то на каком-то сайте я даже читала статью о том, что нужно делать разработчику, чтобы закрепить за собой своё рабочее место. Среди перечисленных пунктов был весьма интересный: вопреки всем правилам клинкода писать код так, чтобы его смог в дальнейшем разобрать только тот, кто его и написал. Этот пункт весьма отдаленно граничит с тем, что имел в виду автор книги (ведь он упоминал сложность написанного для других ввиду того, что они далеки от этой области), но суть близка.

ВЦКП в облаках

"Значит ли это, что софтостроительные фирмы теперь, как утверждают некоторые маркетологи, «производят услуги»?" Прочитав этот вопрос в книге и учитывая достаточно смелый характер изложения материала автором, боялась прочитать дальше положительный ответ. Но не тут-то было. Авторское "Разумеется, нет." весьма и весьма порадовало. Хочется всё-таки и дальше верить, что софтостроение останется "мощным двигателем" решения проблем, грубо говоря "производителем" основы для услуги, но не скатится до просто "услуги".

От CORBA к SOA

«По умолчанию новые технологии олицетворяют прогресс."

Если в данном примере сопоставить "новую технологию" с "идеей", то не все же идеи считаются по умолчанию гениальными. Хотя, конечно, наличие хоть каких-нибудь идей куда лучше их отсутствия.

"…отказаться от «новых» технологий рядовым разработчикам непросто, особенно работающим в сфере обслуживания под руководством менеджеров среднего звена с далёким от технического образованием, оперирующих понятиями освоения и расширения бюджета и массовости рынка специалистов, а не технологической эффективностью." Меня давно интересует вопрос, почему обычными простыми разработчиками командуют люди, далекие от технических понятий. Может проблема не в том, что "начальники" не могут организовать рабочий процесс, а в том, что не находятся такие люди, которые сделают это хорошо?

"Экономика потребления обязана крутиться, даже если в ней перемалываются миллиардные бюджеты бесполезных трат на модернизацию, переделку и переобучение." Я считаю, что если запущен механизм "разработки прогресса", то стоит исключить такие слова, как "бесполезная трата". Это настолько творческий процесс, что никогда не знаешь, что окажется на самом деле полезным.

NET

"Выбор стоит между «изучать новые возможности» и «решать задачи заказчиков». А если изучать, то как не ошибиться с перспективой оказаться у разбитого корыта через пару лет." Наверно, одна из самых актуальных и обсуждаемых тем, так что же всё-таки актуально для изучения. Никогда не угадаешь, что будет перспективным через пару лет.

Office 2007

Изменение интерфейса программ Microsoft Office – отличный показатель того, как люди быстро привыкают к старому.

Постскриптум

В последнее время весьма трудно написать какой-то отзыв или найти то, к чему можно было придраться или не согласиться. Но вот эта часть книги была особенно сложной в том плане, что половина главы занимала история автора ситуации из жизни, а вторая – материал неизменный, неподлежащий обсуждению, я имею в виду теорию (уровни и слои АИС).
На спец. лабораторных мы также обсуждали причины возникающей сложности в написании конспекта с каждым разом. Пришли к выводу, что "техническая неподкованность" во многих аспектах дает о себе знать. Ну и, конечно же, отсутствие такого большого опыта по темам, освещаемым в книге. Да и как можно оспаривать информацию, которая уже устоялась? Стоит только принять к сведению и согласиться с мнением автора, хоть оно и бывает иногда немного резким.

Ultima-S – КИС из коробки

«Никто из веб-разработчиков пока не выходит за рамки postback-ов, динамическое обновление форм без перегрузки всей страницы, предоставляемое Ultima-S, в AJAX появится через 10 лет. А изначально поддерживаемые трёхмерные, стиля Excel, сетки-таблицы с закладками в динамических формах до сих пор доступны лишь в сторонних компонентах…" Это только лишний раз подчеркивает, как быстро и с пользой развиваются технологии.

"Следуйте совету Джобса: «Обычные художники заимствуют – великие воруют»." Никогда не понимала, такого мышления. Хочешь сделать что-то? Что-то действительно хорошее? Воплощай те идеи, которые ещё не реализованы. Будь "новым" в какой-то сфере.

Нешаблонное мышление

"Неокрепшему за недостатком практики уму проектировщика надо научиться самому находить такие решения и пути к ним. Для чего гораздо эффективнее первое время «изобретать велосипеды», нежели сразу смотреть на готовые чужие. На чужие надо смотреть, когда придуман хотя бы один собственный, чтобы понять, насколько он несовершенен, и выяснить, каким же путём можно было бы прийти к лучшим образцам велосипедов данной модели…" Гораздо более эффективен "метод проб и ошибок", когда ты самостоятельно пытаешься что-то сделать. А смотреть за чужими "велосипедами" помогает совершенствовать своё, стремиться к чему-то лучшему.

"Ещё недавно привычка шаблонно мыслить считалась в инженерном сообществе признанием ограниченности специалиста, наиболее пригодного для решения типовых задач с 9 утра до 6 вечера. Теперь не стесняются писать целые книжки о том, как научиться шаблонному мышлению…" Здесь, наверно, имеется в виду "мыслить шаблонно" во время организации. То есть гораздо удобнее, когда организация какой-либо задачи построена логически, имеет свою структуру, пусть и повторяющуюся, зато понятную и проверенную. А вот что касается идей, именно здесь и приветствуются инновации и креатив.

Думать головой

"Потому что думать надо не шаблонами, а головой." Вот эта строчка как раз и подтверждает мои рассуждения о мышлении, изложенные выше.

Про сборку мусора и агрегацию

"По словам М. Донского, наличие в некоторых языках механизма сборки мусора, является примером отказа от самой идеи справиться с этими проблемами и молчаливым признанием возможности присутствия в среде объектов, не принадлежащих ни подпрограммам, ни другим объектам." На самом деле по сборщику мусора столько различных мнений и весьма противоречивых. Ведь с одной стороны огромный плюс в простоте, в том, что программисту не приходится задумываться о "ненужных и забытых" объектах (особенно, если код программы очень велик). Но и минусов хватает. Никто же не знает принцип работы этого механизма. Пока что, наверно, такое мнение будет перевешивать: "Наиболее очевидное преимущество – программисту не надо заботиться об освобожден.

Журнал хозяйственных операций

Вся глава – завуалированная пропаганда синхронизации. Бесспорно, это очень важная часть (процесс) программирования. После прочтения в книгах о синхронизации, могу сказать, что весьма печально, что применение этого процесса описывается только в экономической сфере. Всё-таки мне было бы интереснее прочитать об использовании в других областях, а также о пагубном влиянии синхронизации, если такое вообще имеется.

UML и птолемеевские системы

"Из положительных моментов UML и поддерживающих его сред необходимо отметить, что в ситуации, когда задача складывается, как мозаика, из обрывочных сведений представителей заказчика, прецеденты помогут хоть как-то формализовать и зафиксировать неиссякаемый поток противоречивых требований. "

"Наиболее проработанная часть UML – диаграмма классов – при определённых усилиях и дисциплине программистов, конечно, поможет вам автоматизировать генерацию части рутинного кода приложения."

Учитывая то, что "рисование кода" итак достаточно полезно в программировании (я имею в виду изображение задачи на листочке. Ведь так гораздо удобнее увидеть, что требуется, и структурировать дальнейшую работу.), то UML-диаграммы точно получат положительные отзывы.

«Оптисток», или распределённый анализ данных

Весьма сложно как-то комментировать прочитанную информацию, в которой либо рассказывается история решения конкретной проблемы, например, упрощение функционала, либо просто обсуждается какая-то технология (в данном случае, .NET) или проект ("Оптисток"). Сложность заключается в том, что сами мы с такими проблемами еще не сталкивались в силу небольшого опыта. Поэтому как-то комментировать, а уж тем более, давать оценку ситуации или совет слегка затруднительно.

Архитектура сокрытия проблем

«Последние годы я вижу тотальное падение компетенции в области баз данных. DBA, проснитесь!» Это как стране не хватает хороших экономистов, так и IT-компании администраторов базы данных (как, собственно говоря, и самих знаний вот этих БД).

Code revision, или Коза кричала

"ревизия кода, несомненно, весьма полезная процедура, но как минимум при двух условиях:

эта процедура регулярная и запускается с момента написания самых первых тысяч строк;" Это безусловно так. Ведь гораздо проще уследить за ходом и идеей программы при малом количестве строк в коде. И ошибки гораздо проще искать. В то время, как в большой программе риск упустить их гораздо больше.

Наживулька или гибкость?

Если бы в водопадной методологии можно было бы не только "спускаться", но и "подниматься" по этапам жизненного цикла, то такая методология вполне могла бы стать универсальной и наиболее часто используемой.

"Необходимо отличать спиральную модель от итеративной. Спиральная модель сходится в точку «система готова», итеративная модель в общем случае не сходится, но обеспечивает реализацию всё новых и новых требований." Когда нами на спецкурсе обсуждались различные методологии жизненного цикла IT-проекта, мы не смогли найти различия в итеративной и спиральной методологии. Более того, преподаватель сказал, что мало кто в принципе его понимает. А надо было всего лишь открыть книжку Тарасова.

"С другой стороны, требования к уровню программиста ограничиваются знанием конкретных технологий кодирования, стандартных фреймворков, «умением разбираться в чужом коде» и «умением работать в команде», уже упоминавшимся в словаре для начинающего соискателя. Способность решать олимпиадные задачки здесь от вас не требуется. Скорее, наоборот, будет помехой." Надо учиться искать как можно простое решение поставленной задачи. Если вы не можете такое найти, то, скорее всего, вы решаете неправильно или нерационально.

Остановиться и оглянуться

На протяжении заданного куска текста автор рассказывал о приложении, состоящем из набора слоёв трёхзвенной архитектуры на основе веб-служб. То есть прочитанный материал воспринимается просто как история, констатация факта, а не как материал-почва для дискуссий и рассуждений. В общем-то идея обсуждения книги и сама книга достаточно неплохи, вот только автор зачастую уходит куда-то вглубь материла настолько, что уловить суть происходящего способен только человек, непосредственно имеющий или имевший дело с рассказанным.

Cherchez le bug, или Программирование по-французски

"Определить, что критический порог пройден, несложно, для этого у меня есть один простой признак: «Ты изменил чуть-чуть программу в одном месте, но вдруг появилась ошибка в другом, причём даже автор этого самого места не может сразу понять, в чем же дело»." А как же тот факт, при котором принято считать, что правильно написанная программа должна считаться универсальной? И если "вылетает" в каком-то одном месте, значит программист не учел какой-то случай.

Говорят, что ошибаются все, но сложно представить, что на это способно "такое богатое и «**баг**атое» учреждение, как Microsoft. Хотя было бы забавно взглянуть на код и при обнаружении ошибки с довольным лицом сказать: "Господин Гейтс, а ваши программисты накосячили тут".

"Если же программист уже после обнаружения ошибки говорит: «А у меня она в этом месте работает…», лучше сразу его уволить, чтобы не мучился." А знаете, это мне напомнило моего старосту, у которого, каждого, кто не может решить примеры с интегралами, необходимо отчислить.

Простые правила чтения специальной литературы

"Технические книжки – это не бессмертные произведения графа Толстого. К ним должен быть суровый, сугубо индивидуальный и безжалостный подход без какой-либо оглядки на чужое мнение."

Литература и программное обеспечение

"Хотя софтостроение довольно часто сравнивают со строительством домов, видимо, желая поскорее свести роль программистов к укладке готовых кирпичей и тем самым закрыть надоевшую проблему огромного числа просроченных или неудачных проектов, не бойтесь аналогий с писательским трудом."

Ровдо Дарья

О нашей профессии

Специализация:

Программист = алгоритмизация и кодирование

Программист минус алгоритмизация = кодировщик

Программист минус кодирование = постановщик задачи

Нельзя «сделать красиво», если относиться к работе исключительно утилитарно и шаблонно. Но и нельзя «сделать красиво», если рассматривать софтостроение лишь как искусство и средство самовыражения. Невозможно обойтись без знаний технологий производства и хороших ремесленных навыков.

Программирование нельзя целиком причислить ни к искусству, ни к ремеслу, ни к науке. Софтостроение на текущий момент – эклектичный

сплав технологий, которые могут быть использованы как профессионалами технического творчества, так и профессионалами массового производства по

шаблонам и прецедентам.

Софтостроение — продукт и услуги:

На одного производителя тиражируемого продукта разной степени серийности – от массовых брендов до малотиражных специализированных, приходится почти десяток поставщиков услуг, крутящихся вокруг этих продуктов, и разработчиков заказных программных систем.

Круговорот:

Вовлечение в сферу услуг исключенных из производственных цепочек людей.

Конкуренция по себестоимости разработки:

Если в производстве она тесно связана со снижением издержек, то в сфере услуг на первый план выходят доверительные отношения между заказчиком и подрядчиком, снижающие риски.

Огромное количество программистов:

Немало специалистов высокой квалификации уходят в экспертизу и консалтинг, где проводят аудит, обучение, «натаскивание» и эпизодически «вправляют мозги» разным группам разработчиков из числа переквалифицировавшихся. которые курсы переквалификации выдать не могут. Другой доступный вариант – специализация на предметных областях.

CV:

• Краткость – сестра таланта.

• Кто ясно мыслит, тот ясно излагает (ёмкие и краткие формулировки).

• Не фантазируйте.

• Не врите.

• Если соискание касается технического профиля, в каждом описании опыта работы упирайте на технологии, если управленческого – на периметр ответственности, если аналитического – на разнообразие опыта и широту кругозора.

• Не делайте ошибок (имеется в виду грамматических, пунктуационных и т.д.).

• Будьте готовы , что далее первой страницы ваше резюме читать не станут.

Мотивация и стимуляция:

Мотивация – внутренний механизм. Чтобы управлять им, необходимо залезать в психику. Напротив, стимуляция – это внешний механизм. Он основан на выявлении мотивов и последующем их поощрении или подавлении.

Управлять мотивацией, то есть целенаправленно изменять психологию и выстраивать набор стимулов – это «две большие разницы». Первое, по сути, требует изменения самих людей, второе – это использование имеющихся у них мотивов. Мотивировать же можно только свои поступки, но никак не образ действия окружающих.

С чем не согласна:

Большего всего, пожалуй, я не согласна со следующим высказыванием:

«Не будет иметь большого значения то, что ты можешь сделать хороший дизайн, если за тобой на интервью придёт дилетант, заучивший десять известных работодателю «паттернов», 200 классов фреймворка и просящий за это в 2 раза меньше денег». Возможно, я слишком оптимистична, но я думаю, что много компаний сейчас на собеседовании спрашивают именно про те задачи, которые вы должны уметь решать. Например, если уже речь о дизайне, почти наверняка спросят, как лучше сделать такую или такую разметку. По такого рода вопросам и определяется ваш настоящий уровень, это важнее чем то, сколько атрибутов вы заучили. Я уверена, что работодатель думает так же.

«Словарик ключевых фраз, часто присутствующих в объявлении о вакансиях» тоже вызывает у меня некоторые сомнения. Возможно, что-то в пояснениях и является правдой, однако много всего явно преувеличено :)

С чем согласна:

В целом, я согласна во всем кроме выше перечисленного :)
Однако хотелось бы выделить отдельно некоторые моменты.

«Если вы не работаете на производстве у одного из поставщиков тиражируемого программного обеспечения, то взаимодействия типа «человек –человек» становятся необходимым и важным элементом повседневной работы, если только вы не предполагаете всю жизнь провести в кодировании чужих спецификаций, не всегда толковых и формализованных». Не раз уже было оговорено, что сейчас программист - уже не просто человек, которому для эффективной работы нужен только компьютер. Умение выстраивать отношения очень важно.

Также очень понравились основные принципы хорошего резюме. Не то, что бы они были чем-то новым, нет. Но эти советы, на мой взгляд, действительно являются полезными, еще и расписаны очень доступным языком.

И, конечно же, рассказ из реальной жизни в конце главы. Думаю, со многими хорошими программистами случалось нечто похожее, отличное завершение темы.

Технологии

Можно ли конструировать программы как аппаратуру?

В софтостроении использовать конечно-автоматную модель для программного компонента можно при двух основных условиях:

• Программисту не забыли объяснить эту теорию ещё в вузе.

• Количество состояний обозримо: они, как и переходы, достаточно легко определяются и формализуются. На практике количество состояний даже несложного модуля запредельно велико, поэтому программист использует их объединения в группы и применяет различные эвристики для обеспечения желаемого результата на выходе при заданном входе.

Вдобавок к модульному тесту необходимо программировать тест производительности, который тем не менее не гарантирует время отклика, а только позволяет определить его ожидаемое значение при некоторых условиях. Таким образом, собрав из кучи микросхем устройство, мы уверены, что оно будет работать:

• согласно таблицам истинности;

• с заданной тактовой частотой.

Собрав же из компонентов программу, мы можем только:

• приблизительно и с некоторой вероятностью оценивать время отклика на выходе;

• в большинстве случаев ограничиться выборочным тестированием, забыв о полноте.

Безысходное программирование

Безысходное программирование – это программирование без «исходников». То есть мы пишем свой код, не имея исходных текстов используемой подпрограммы, класса, компонента и т. п.

Когда необходимо обеспечить гарантированную работу приложения, включающего в себя сторонние библиотеки или компоненты, то, не имея доступа к их исходному коду, вы остаётесь один на один с «чёрным ящиком». Даже покрыв их тестами, близкими к параноидальным, вы не сможете понять всю внутреннюю логику работы и предусмотреть адекватную реакцию системы на нестандартные ситуации.

Частным, но частым случаем безысходного программирования является софтостроение без использования системы управления исходным кодом (revision control system), позволяющей архивировать и отслеживать все его изменения.

Эволюция аппаратуры и скорость разработки

Производительность «железа» возросла на порядки, почти упёршись в физические ограничения миниатюризации полупроводников и скорость света. Стоимость тоже на порядки, но снизилась. Увеличилась надёжность, развилась инфраструктура, особенно сетевая. Параллелизация вычислений пошла в массы на плечах многоядерных процессоров. Прежними остались лишь принципы, заложенные ещё в 1930-х годах и названные, согласно месту, Гарвардской и Принстонской архитектурами ЭВМ. Вчерашний студент теперь пишет не на ассемблере и C, а на Java, будучи уверенным в принципиальной новизне ситуации, не всегда осознавая, что изменилось только количество герц тактовой частоты и байтов запоминающих устройств.

Массовые технологии, доступные шести миллионам программистов, являются универсальными, то есть могут быть использованы для разработки большинства типов программ, пакетов и систем. Поэтому важным элементом бизнеса становится не столько сокращение срока разработки, сколько максимизация использования стандартных сред, компонентов и фреймворков. И хотя по срокам, бюджету и количеству разработчиков владеющие специализированными технологиями выигрывают у бригады «универсалов», но возникающие при этом риски могут свести к минимуму весь выигрыш.

Конечно, специализированные средства разработки всегда обеспечат преимущества по сравнению с универсальными. Тем не менее основная разработка по-прежнему будет идти на весьма ограниченном наборе универсальных сред и фреймворков, выталкивая специализированную в ниши, где сроки и производительность являются наиболее важными.

Количество работающих в софтостроении женщин росло до начала 1990-х годов, после чего резко пошло на убыль. Такая тенденция иллюстрирует факт ухода технологий софтостроения от специализированных сред, не требующих работы на далёком от решаемой прикладной задачи уровне математических абстракций, в которых прекрасный пол почему-то считается менее способным разбираться.

Впоследствии мне не раз приходилось видеть исходники программисток на вполне себе объектно-ориентированном Delphi/C++Builder. В прикладном коде никакого объектного подхода, конечно, не было, всё ограничивалось компоновкой экранных форм стандартными элементами среды и написанием обработчиков событий в процедурном стиле. Разумеется, это говорит не о «плохости» ООП, а о высоком уровне компетенции, необходимом, чтобы эта технология давала осязаемые преимущества. Тогда как цель прикладника – побыстрее собрать работающее решение для заказчика.

Диалог о производительности

Вместо прямого пути с настраиваемым источником данных ради приобщения к действу ещё одного «выпускника курсов» заказчик выбрал локальный расчёт с последующей перекачкой данных. В итоге используемое дисковое пространство удваивается, время увеличивается.

С чем не согласна:

«Частным случаем безысходного программирования является софтостроение без использования системы управления исходным кодом» - я так понимаю, что тут слово «безысходное» имело свое прямое значение, а не заявленное ранее в данном пункте. Иначе я не просто не согласна с этим утверждением, а вообще его не понимаю.

«Вчерашний студент теперь пишет не на ассемблере и C, а на Java, будучи уверенным в принципиальной новизне ситуации, не всегда осознавая, что изменилось только количество герц тактовой частоты и байтов запоминающих устройств», «и хотя по срокам, бюджету и количеству разработчиков владеющие специализированными технологиями выигрывают у бригады «универсалов», но возникающие при этом риски могут свести к минимуму весь выигрыш». Во-первых, начинает складываться впечатление, что автор книги — пессимист. Во-вторых, на мой взгляд, на данный момент как раз ситуация, в которой «универсалы» ценнее, сменяется той, в которой если ты хочешь быть более востребованным, стоит углубить свои знания в некоторой определенной узкой области.

«Впоследствии мне не раз приходилось видеть исходники программисток на вполне себе объектно-ориентированном Delphi/C++Builder» - честно говоря, немного обидно это читать :). Уверена, что на этапе перехода к этим языкам программирования, у программистов мужского пола исходники тоже

недалеко ушли. Опять же, если говорить о настоящем моменте времени, программисток не так уж и мало.

С чем согласна:

В данной теме я полностью согласилась только с параграфом «Можно ли конструировать программы как аппаратуру?». Даже добавить к этому нечего, все четко расписано, что, как и почему, все очень логично.

О карманный монстрах

Лет 10–15 назад можно было бы взять на выбор Delphi/C++ Builder, PowerBuilder, Visual Basic, FoxPro, лёгкую клиент-серверную СУБД и сделать приложение за 3–5 дней с написанием каких-то сотен строк прикладного кода. Внесение изменений типа «добавления атрибута к сущности» вместе воссозданием инсталлятора и скрипта обновления базы данных занимало час-два.

В 2009 году приложение было сделано на платформе. NET в трёхзвенной архитектуре: сервер приложений на базе WCF, Entity Framework, СУБД SQL Server 2005 и клиент в виде подключаемого модуля (add-in) к Office 2007 на WinForms. Спасибо, что не на WPF.

Приложение занимает примерно 20 тысяч строк на C#, из них более половины являются техническими: слой объектов доступа к данным, прокси классов для

WCF и прочая начинка. Конфигурационный файл для WCF-сервера – 300 строк XML. Это больше, чем нужно написать, например, Delphi-кода для логики отображения форм во всем приложении.

• менеджеру, в соответствии с корпоративным стандартом, необходимо было использовать только платформы и средства Microsoft;

• программист не имел опыта разработки вне шаблонов многозвенной архитектуры и проекций объектов на реляционную СУБД, поэтому не стал рисковать.

ASP.NET и браузеры

Привыкшему к интерактивности полноценных приложений пользователю одних лишь кнопок не хватает. Тогда и в браузеры (то есть на стороне клиента) тоже

включили поддержку скриптовых языков. В итоге исходная веб-страница, ранее содержавшая только разметку гипертекста, стала включать в себя скрипты для выполнения вначале на сервере, а затем и на клиенте. Можете представить, какова была эта «лапша» на сколько-нибудь сложной странице ASP. Многие сотни строк каши из HTML, VBScript и клиентского JavaScript.

Последующая эволюция технологии была посвящена борьбе с этой лапшой, чтобы программный код мог развиваться и поддерживаться в большем объёме и не только его непосредственными авторами. На другом фронте бои шли за отделение данных от их представления на страницах, чтобы красивую обёртку рисовали профессиональные дизайнеры-графики, не являющиеся программистами.

Легко проследить даже на простом примере, что для программиста помимо решения собственно прикладной задачи находится уйма забот. Основной целью такой дополнительной головной боли является платформенная независимость клиентской части приложения и максимально облегчённое развёртывание так называемого «тонкого» клиента, которым является веб-браузер.

Достаточно быстро выяснилось, что разработка приложения, корректно работающего хотя бы под двумя типами браузеров (Internet Explorer, Netscape и

впоследствии Mozilla) – задача не менее сложная, чем написание кода в автономном приложении на базе переносимой оконной подсистемы (Lazarus, C++ и другие). А тестировать нужно не только под разными браузерами, но и под разными операционными системами. С учётом версий браузеров.

Апплеты, Flash и Silverlight

Появившись в 1995 году, технология Java сразу пошла на штурм рабочих мест и персональных компьютеров пользователей в локальных и глобальных сетях. Наступление проводилось в двух направлениях: полноценные «настольные» (desktop) приложения и так называемые апплеты 24, то есть приложения, име-

ющие ограничения среды исполнения типа «песочница» (sandbox). Например, апплет не мог обращаться к дискам компьютера.

Несмотря на значительные маркетинговые усилия корпорации Sun, результаты к концу 1990-х годов оказались неутешительны: на основной платформе пользователей – персональных компьютерах – среда исполнения Java была редким гостем, сами приложения можно было сосчитать по пальцам одной руки (навскидку вспоминается только Star Office), веб-сайтов, поддерживавших апплеты, было исчезающе мало, а

настойчивые просьбы с их страниц скачать и установить 20 мегабайтов исполняемого кода для просмотра информации выглядели издевательством при существовавших тогда скоростях и ограничениях трафика.

Причины:

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

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

Тем не менее необходимость в кросс-платформенных богатых интерактивными возможностями интернет-приложениях27 никуда не исчезла, поскольку браузеры, нашпигованные скриптовой начинкой, обладали ещё большими техническими ограничениями и низким быстродействием даже по сравнению с апплетами. Эта ниша к началу 2000-х годов оказалась плотно занятой Flash-приложениями, специализирующимися на отображении мультимедийного содержания. Учтя ошибки Java, разработчики из Macromedia сделали инсталляцию среды исполнения максимально лёгкой в загрузке и простой в установке.

К решению проблемы подключилась Microsoft. Первым «блином» в 2005 году стала технология ClickOnce развёртывания полноценных WinForms-приложений.

По-прежнему клиентское рабочее место требовало предварительно установки среды исполнения. NET версии 2. Но развёртывание и автоматическое обновление приложения и его компонентов было полностью автоматизировано. Первоначально пользователь, не имеющий прав локального администратора,

устанавливал необходимую программу, просто щёлкнув по ссылке в браузере, далее запуская её с рабочего стола или из меню. Sun отреагировала молниеносно, добавив аналогичную возможность под названием Java Web Start.

Но «блин» всё-таки вышел комом. Имея полную возможность предустановить

среду. NET 2 на все рабочие места вместе с очередным пакетом обновлений, Microsoft не решилась на такой шаг, тем самым фактически похоронив массовое использование новой технологии разработчиками, имевшими неосторожность надеяться на помощь корпорации в развёртывании тяжёлых клиентских приложений.

Возможно, одной из причин стала потеря интереса Microsoft к WinForms, чьё развитие было заморожено, и переход в. NET 3 к более общей технологии построения пользовательских

интерфейсов WPF 29, отличающейся универсальностью и большей трудоёмкостью в прикладной разработке, но позволяющей полностью разделить труд программистов и дизайнеров, что имело смысл в достаточно больших и специализированных проектах.

Побочным продуктом WPF стал Silverlight. По сути, это реинкарнация Java-апплетов, но в 2007 году, спустя более 10 лет, и в среде. NET. Кроме того Silverlight должен был по замыслу авторов составить конкуренцию Flash в области мультимедийных интернет-приложений.

В отличие от WPF, Silverlight вызвал больший энтузиазм разработчиков корпоративных приложений. Во-первых, для развёртывания не требовалась вся

среда. NET целиком, достаточно было установить её часть, размер дистрибутива которой составлял всего порядка 5 мегабайтов. Поэтому на очередные обещания Microsoft предустановить. NET 3 можно было не полагаться, тем более при уже анонсированном. NET 3.5. Во-вторых, приложение можно было запускать не только в окне браузера, но и автономно.

Прежде всего насторожили меня новости про отсутствие в Silverlight отличных от юникода 30 кодировок. Их нет в константах, а Encoding.GetEncoding

(1251) выдаёт ошибку. Как корректно импортировать в приложение ASCII 31-файл? Никак. Из этого вытекала невозможность полноценной работы приложения с обыкновенным текстовым файлом данных, вроде

CSV (comma separated values). Прямой доступ к базам данных также отсутствовал. Можно было пойти окольными путями через COM interops и ADO, но для этого требовались очень серьёзные поводы.

Silverlight вырос до вполне взрослой версии 4, уже давно вышла Visual Studio 2010, где встроена поддержка разработки приложений под него. Но зададимся вопросом: «Может ли пользователь установить себе Silverlight-приложение, не будучи администратором на своем компьютере?» Ответ, мягко говоря, разочаровывающий: «Нет, не может».

При соответствующих ограничениях развёртывания выбор по-прежнему лежит между веб-браузером или условным Delphi, хочешь ты этого или нет...

С чем не согласна:

В первых двух пунктах «О карманных монстрах» и «ASP.NET и браузеры», по сути, говорят о том, что более старые технологии выигрывают у более новых в сложности написания кода, в его количестве. В принципе, это действительно так. Однако это преподнесено таким образом, что как будто ставятся под сомнения плюсы этих самых новых технологий, с чем я согласиться никак не могу. Я сомневаюсь, что приложение на Delphi сейчас может тягаться с симпатичными и функциональными веб-приложениями.

С чем согласна:

Честно говоря, я не могу сделать ничего другого с пунктом «Апплеты, Flash и Silverlight», кроме как согласиться. Это информация для меня является новой, я не могу быть с ней не согласна :)

ООП – неизменно стабильный результат

В начале широкой популяризации ООП, происходившей в основном за счёт языка C++, одним из главных доводов был следующий: «ООП позволяет увеличить количество кода, которое может написать и сопровождать один среднестатистический программист». Как только «главным программистом» стал «коллективный разум» муравейника, проблема мгновенно всплыла, порождая

Ад Паттернов, Чистилище нескончаемого рефакторинга и модульных тестов, недосягаемый Рай генерации по моделям кода безлюдного Ада. «Ад 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 сложных и многоуровневых конструкций в стиле «новый адаптер вызывает старый» по мере так называемого эволюционного развития системы;

• лоскутная интеграция существующих систем и создание поверх них новых слоёв API.

Последствия от создания Ада Паттернов ужасны не столько невозможностью быстро разобраться в чужом коде, сколько наличием многих неявных зависимостей. Например, в рамках относительно автономного проекта мне пришлось интегрироваться с общим для нескольких групп фреймворком ради вызова единственной функции авторизации пользователя: передаёшь ей имя и пароль, в ответ «да/нет». Этот вызов повлёк за собой необходимость явного включения в. NET-приложение пяти сборок. После компиляции эти пять сборок притащили за собой ещё более 30, большая часть из которых обладала совершенно не относящимися к безопасности названиями, вроде XsltTransform. В результате объём дистрибутива для развёртывания вырос ещё на сотню мегабайтов и почти на 40 файлов. Вот тебе и вызвал функцию…

В обычной ситуации оказывается, что C ++, действительно увеличивавший личную продуктивность Страуструпа и его коллег, начинает тормозить производительность большого софтостроительного цеха где-нибудь в жарком субтропическом опенспейсе площадью в гектар. Помимо общих проблем интеграции, на C++ достаточно просто «выстрелить себе в ногу», и человеческий фактор быстро становится ключевым риском проекта.

Появились новые C-подобные языки: сначала Java, а чуть позже и C#. Они резко снизили порог входа за счёт увеличения безопасности программирования, ранее связанной прежде всего с ручным управлением памятью.

Оказалось, что ООП очень требовательно к проектированию, так и оставшемуся сложным и недостаточно формализуемым процессом.

Реальную отдачу от ООП вы получите, только создав достаточно хорошие модели предметных областей. То есть тот самый минимально необходимый словарь и язык вашей системы для выражения её потребностей, доступный не только современным Пушкиным от программирования. Модели будут настолько простыми и ясными, что их реализация не погрузит команду в инфернальные нагромождения шаблонных конструкций и непрерывный рефакторинг, а фреймворки будут легко использоваться прикладными программистами без помощи их авторов. Правда, тогда неизбежно встанет вопрос о необходимости использования ООП…

ORM, или объектно- реляционный проектор

Сокрытие базы данных, или Как скрестить ежа с ужом

Именно реляционным СУБД удалось в 1980-х годах освободить программистов от знания ненужных деталей организации физического хранения данных, отгородиться от них структурами логического уровня и стандартизованным языком SQL для доступа к информации. Также оказалось, что большинство форматов данных, которыми оперируют программы, хорошо ложатся на модель двумерных таблиц и связей между ними. Эти два фактора предопределили успех реляционных СУБД, а в качестве поощрительной премии сообщество получило строгую математическую теорию в основании технологии.

На практике сложилась ситуация, когда программы пишутся в основном с использованием ООП, тогда как данные хранятся в реляционных БД. Не касаясь пока вопроса целесообразности такого скрещивания «ежа с ужом», примем ситуацию как данность, из которой следует необходимость отображения (проецирования) объектов на реляционные структуры и обратно.

Компонент программной системы, реализующий отображение, называется ORM, или объектно-реляционный проектор – ОРП. Полноценный ORM может быть весьма нетривиальным компонентом, по сложности превосходящим остальную систему. Поэтому, хотя многие разработчики с успехом пользуются своими собственными частными реализациями, в отрасли за последние 10 лет появилось несколько широко используемых фреймворков, выполняющих в том числе и задачу проекции.

SQL – высокоуровневый декларативный специализированный язык четвёртого поколения, в отличие от того же Java или C#, по-прежнему относящихся к третьему поколению языков императивных. Единственный оператор SQL на три десятка строк, выполняющий нечто посложнее выборки по ключу, потребует для достижения того же результата в разы, если не на порядок, больше строк на C#. Такая ситуация приводит разработчиков ORM к необходимости создавать собственный SQL-подобный язык для манипуляции объектами и уже его транслировать в сиквел (см. HQL). Или использовать непосредственно SQL с динамическим преобразованием результата в коллекцию объектов.

Как обычно используют ORM

На софтостроительных презентациях часто рисуют красивые схемы по разделению слоёв представления, бизнес-логики и хранимых данных. Голубая мечта начинающего программиста – использовать только одну среду и язык для разработки всех слоёв и забыть про необходимость знаний реляционных СУБД, сведя их назначение к некоей «интеллектуальной файловой системе».

• Вначале происходит выбор ORM-фреймворка для отображения. Уже на этом этапе выясняется, что с теорией и стандартами дело обстоит плохо. Впору на- сторожиться бы, но презентация, показывающая, как за 10 минут создать основу приложения типа записной книжки контактов, очаровывает. Решено!

• Начинаем реализовывать модель предметной области. Добавляем классы, свойства, связи. Генерируем структуру базы данных или подключаемся к существующей. Строим интерфейс управления объектами типа CRUD. Все достаточно просто и на первый взгляд кажется вполне сравнимым с манипуляциями над DataSet – тем, кто о них знает, конечно – ведь не все подозревают о существовании табличных форм жизни данных в приложении за пределами сеток отображения DBGrid.

Внезапно оказывается, что собственный язык запросов генерирует далеко не самый оптимальный SQL.

Триггер как идеальная концепция для NHibernate

Обычно разработчикам баз данных я рекомендую избегать необоснованного использования триггеров. Потому что их сложнее программировать и отлаживать. Оставаясь скрытыми в потоке управления, они напрямую влияют на производительность и могут давать неожиданные побочные эффекты.

Однако стоит посмотреть на слой домена, живущего под управлением NHibernate, как становится ясно, что триггер в СУБД – это достаточно простая и хорошо документированная технология. В то время как NHibernate предлагает прикладному разработчику целый зоопарк триггероподобных решений.

Эмпирика

В качестве одной из метрик оценки качества реализации приложений корпоративной информационной системы можно принять соотношение числа таблиц в базе данных к числу тысяч строк кода программ без учёта тестов. Чем меньше кода, тем лучше, это понятно. Например, качественная реализация слоя хранения использует DRI и прочую декларативность на уровне метаданных вместо императивного кодирования такой логики.

• Соотношение 1 к 1–2 примерно соответствует тому порогу, за которым начинается так называемый «плохой код».

• 1 к 3–4 – следует серьёзно заняться изучением вопроса переделки частей системы.

• 1 к 5 и более – надеемся, что случай нестандартный (сложные алгоритмы, распределенные вычисления, базовые подсистемы и компоненты реализуются самим разработчиком), либо «врач сказал – в морг».

С чем не согласна:

Так получилось, что в этой части книги мне не с чем не согласиться.

С чем согласна:

Взгляд на ООП, описанный в книге, я полностью поддерживаю. То есть как, полностью. Я уверена, что ООП, конечно, вещь полезная, в некоторых случаях работать с ней приятно, ощущаешь удовлетворение от того, что решение проблемы такое элегантное. Однако зачастую оно оказывается и слишком сложным. Это огромное количество интерфейсов, эти наследования, иногда смотришь и думаешь, зачем все это, ради какой-то небольшой функции? Может, самой ее реализовать по-быстрому? Очень хороший пример был из жизни автора с функциией авторизации.

SQL vs собственный язык запросов ORM-фреймворков.

Актуально как никогда. Перед тем, как прочитать данную часть книги, я потратила 3 часа на написание сложного запроса, используя Criteria в Hibernate. 2,5 часа из них я мысленно ругалась. Этот же запрос я написала днем ранее в 3 строки на SQL, но интересно же было узнать, что там придумал Hibernate, чтобы было удобнее разработчикам. Итог — 30 строк кода. И еще при этом вообще непонятно, как реально выглядит запрос в базу данных. Подозреваю, что я буду еще раз писать с использованием Criteria только если меня очень сильно попросят.

Вообще мне кажется, что все-таки собственные языки запросов придумали для простых случаев. Что объемные запросы даже не очень предусмотрены. Может, подразумевается, что если уж ты хочешь писать что-то сложное, то должен разбираться в сути, а если уж разбираешься в сути, то пиши хранимые процедуры и не полагайся на фреймворк? :)

ВКЦП в облаках

На заре массового внедрения АСУ на предприятиях в 1970-х годах советские управленцы и технические специалисты предвидели многое. А именно, сосредоточение ресурсов в вычислительных центрах, где конечные пользователи будут получать обслуживание по решению своих задач и доступ. Доступ по простым терминалам, в ту эпоху ещё алфавитно-цифровым. Такова была концепция ВЦКП – Вычислительного Центра Коллективного Пользования.

Если частично заменить «большие ЭВМ» на «кластеры из серверов», добавить взращенную за последние 10 лет широкую полосу пропускания общедоступных сетей на «последнем километре» 70, а в качестве терминала использовать веб-браузер или специальное приложение, работающее на

любом персональном вычислительном устройстве, от настольного компьютера до коммуникатора, то суть не изменится. Облако – оно же ВЦКП для корпоративного и даже массового рынка.

Интересно попытаться усмотреть тенденции технологического маятника, резко качнувшегося в 1980–90-х годах в сторону децентрализации и теперь возвращающегося на позиции годов 1970-х в качественно новой ситуации обилия дешёвых терминалов с их избыточными мощностями и общедоступными высокоскоростными каналами связи.

SaaS и манипуляции терминами

По причине развития общедоступных каналов связи работать с программами можно с удалённого терминала, в роли которого выступает множество

устройств: от персонального компьютера до мобильного телефона. Эксплуатацию же программ ведут поставщики услуг в «облаках» – современные ВЦКП. Это и есть «программа как услуга», SaaS.

В схеме «программа как услуга» посредник (ВЦКП) берет на себя эксплуатацию программного продукта, то есть его установку, настройку, содержание, обновление и т. п., предлагая конечному пользователю услугу по доступу к собственно функциональности этой программы. Однако софтостроители продолжают производить продукт, а не услугу. Конечным пользователем для них является ВЦКП в «облаках», оказывающий на базе этого продукта услуги своим клиентам. Программа – это продукт, как и многие другие, например автомобиль, позволяющий на своей основе оказывать такие полезные услуги, как прокат, извоз или транспортировка.

От CORBA к SOA

CORBA – одна из технологий создания сервис-ориентированной архитектуры.

CORBA имела очевидные достоинства, прежде всего это интероперабельность (англ. interoperability – способность различных по архитектуре, аппаратным платформам и средам исполнения систем к взаимодействию) и отраслевая стандартизация не только протоколов (шины), но и самих служб и общих механизмов – facilities. Например, служб имён, транзакций, безопасности, хранения объектов, конкурентного доступа, времени и т. п. Однажды реализованный программистами сервис мог использоваться многими системами без какой-либо дополнительной адаптации и ограничений с их стороны.

Почему же CORBA была оттеснена на обочину? Причин много, но хотелось бы выделить следующие:

• Стандарт поддерживался многими корпорациями, поэтому развитие требовало длительного согласования интересов всех участников. Некоторые из них продвигали свои альтернативы, например Sun Java RMI и Jini, Microsoft DCOM.

• Несмотря на реальную поддержку интероперабельности, множества языков и сред программирования, основным средством разработки оставался C++. Но код на C++, манипулирующий инфраструктурой CORBA и службами, является излишне сложным по сравнению с той же Java или даже Delphi. Практиковавшие подтвердят, остальным будет достаточно взглянуть на примеры в Сети. А Java-программисты не спешили использовать CORBA из-за упомянутых

альтернатив.

• Запоздалая (1999 год), объёмная и весьма сложная спецификация компонентной модели. Java-сообщество к тому времени обладало альтернативой в виде EJB78 с открытыми и коммерческими реализациями.

В начале 2000-х в ИТ-отрасли разразился кризис лопнувшего пузыря интернет-компаний, продвигать сложные и дорогие инфраструктуры, а коммерческие реализации CORBA стоили порядка тысячи

долларов за рабочее место, стало очень трудно. Постепенно из кустов начали выкатывать на сцену рояль веб-сервисов, который из-за проблем с CORBA должен был стать новой платформой интеграции служб и приложений в гетерогенной среде. Концепции быстро придумали многообещающее название COA. Надо сказать, что рояль не стоял без дела и в кустах: разработка веб-служб и развитие XML велись в конце 1990-х и активно продолжились в 2000-х годах. Поскольку это была технология, хотя и весьма базового уровня, но относительно простая, дешёвая и открытая не только на уровне спецификаций, но и в виде множества реализаций.

мом минимальном варианте CORBA.

Веб, точнее, его основа, HTTP – среда без состояния и пользовательских сессий. В общедоступном Интернете такое решение было вызвано соображениями нагрузки, поскольку максимальное число запросов к серверу теоретически равно количеству всех устройств в сети. В корпоративной же системе нагрузку на службу можно (и нужно) рассчитать гораздо точнее. В итоге программистам, не связанным с веб-разработкой для Интернета, приходится восполнять недостаток средств протокола надстройками поверх него костылей, например, постоянно гоняя контекст и состояние в сообщениях или симулируя сессии по тайм-ауту.

В CORBA сессии поддерживались средой без дополнительных усилий. Если в частных случаях возникали вопросы нагрузки на поддержку соединений при большом количестве клиентов, они решались так же просто, как и в среде СУБД: приложение самостоятельно отсоединялось от сервера, выполнив пакет

необходимых запросов. При желании нетрудно было также организовать и принудительное отсоединение по истечении заданного периода пассивности. Но для корпоративной службы, напомню, речь идёт обычно о десятках и сотнях активных сессий, поддержка которых в большинстве случаев укладывается в ресурсы серверов.

Неприятным следствием отсутствия сессий стала невозможность поддержки транзакций при работе с веб-службами. Для решения проблемы в рамках ООП

необходим шаблон «Единица работы» (unit of work), суть которого в передаче веб-службе сразу всего упорядоченного множества объектов, подлежащих со-

хранению в управляемой сервером транзакции.

Вторым «упрощением» стал переход от понятных прикладному программисту деклараций интерфейсов объектов и служб на языке IDL к WSDL – описаниям, ориентированным, прежде всего, на обработку компьютером.

Третьим «усовершенствованием» стал отказ от автоматической подгрузки связанных объектов в пользу исключительно ручного управления процессом.

Современные заявления о том, что COA не оправдала возложенных на неё надежд, свидетельствуют о том, что и выбранная для неё модель веб-служб не стала решением проблем взаимодействия приложений в корпоративной среде. Ожидает ли нас новое пришествие CORBA в виде облегчённой её версии – покажет время.

Прогресс неотвратим

Наиболее ходовым термином в софтостроении является «новые технологии». По умолчанию новые технологии олицетворяют прогресс. Но всегда ли это так? Даже если не всегда, то в условиях квазимонополий, поделивших рынки корпораций, отказаться от «новых» технологий рядовым разработчикам непросто, особенно работающим в сфере обслуживания под руководством менеджеров среднего звена с далёким от технического образованием, оперирующих понятиями

освоения и расширения бюджета и массовости рынка специалистов, а не технологической эффективностью.

.NET

Цифры версий и релизов фреймворка. NET меняются со скоростью, заметно превышающей сроки отдачи от освоения и внедрения технологий.

Давайте подумаем, кто же выигрывает в этой гонке кроме самой корпорации, пользующейся своим квазимонопольным положением:

• Услуги по сертификации. Прямая выгода.

• Консультанты и преподаватели курсов. Сочетание прямой выгоды с некоторыми убытками за счёт переобучения и очередной сертификации.

• Программисты в целом. Состояние неопределённости. Выбор стоит между «изучать новые возможности» и «решать задачи заказчиков». А если изучать, то как не ошибиться с перспективой оказаться у разбитого корыта через пару лет.

• Разработчики в заказных проектах. Прямые убытки. За пару лет получен опыт работы с технологиями, признанными в новом фреймворке наследуемыми, то есть не подлежащими развитию, хорошо, если поддерживаемыми на уровне исправления критичных ошибок. А ведь всего 2–3 года назад поставщик убеждал, что эти технологии являются перспективными, важными, стратегическими и т. п. Необходимы новые инвестиции в обучение персонала и преодоление появившихся рисков.

• Разработчики продуктов. Косвенные убытки. В предлагаемом рынку продукте важна функциональность и последующая стоимость владения. На чём он написан – личное дело компании-разработчика. Тем не менее заброшенную поставщиком технологию придётся развивать за свой счёт или мигрировать на

новую.

Office 2007

Как известно, Microsoft изменила интерфейс в Office 2007. Вместо привычных меню появились многочисленные закладки лент панелей инструментов с

крупными пиктограммами. По словам Microsoft, это сделано для облегчения работы начинающим пользователям.

Хорошо, возможно, начинающим жить в офисном пакете это полезно или безразлично. А мы, давние пользователи, заканчиваем в нём жить, что ли? Кроме проблем с интерфейсом возникли проблемы открытия файлов в Office 2003. Повторилась ситуация с версией Office 97, когда Microsoft пришлось в срочном

порядке выпускать конвертер для Office 95, позволяющий открывать в нём файлы новых форматов. Разве трудно было предусмотреть возможность выбора между старым и новым интерфейсом, как это было сделано в Windows XP и 7? Зато всего за 30 долларов вам предложат купить программку третьей фирмы Classic Menu, которая возвратит старый интерфейс.

SQL Server

И здесь, начиная с 2005-й версии, наряду с полезными нововведениями проявилась определённая деградация.

Vista

Windows 7 ещё находилась в состоянии «кандидат к выпуску», а представители Microsoft открытым текстом стали предлагать отказаться от покупки своего

флагманского на тот момент продукта – операционной системы Windows Vista и подождать выхода новой версии.

По сути пользователям сообщили, что мы достаточно потренировались на вас и за ваши же деньги, а теперь давайте перейдём собственно к делу. Судьба

Vista была решена – система фактически выброшена в мусорную корзину, так и не успев занять сколь-нибудь значительную долю парка «персоналок» и ноутбуков. Производителям железа была дана отмашка переключиться на Windows 7, корпоративным службам ИТ пришлось в срочном порядке сворачивать проекты по переходу на Vista и ориентироваться на

«семёрку».

О материальном

И немного о материальном, про оборудование. Периферия. Широко распространено мнение о том, что Linux на порядок хуже чем Windows поддерживает всякие периферийные устройства. Однако это не совсем так.

Если брать 32-разрядные версии Windows, то, действительно, нетрудно найти драйвер даже к довольно старому устройству. Но Microsoft уходит с 32-разрядных платформ на десктопах и ноутбуках, навязывая предустановленные 64-разрядные версии своих операционных систем. Некоторые приложения всё чаще требуют 64-разрядной среды, например, для обработки больших объёмов данных или видео высокой чёткости. Тут-то и выясняется, что по уровню поддержки периферии 64-разрядные Windows отрезают вам путь к использованию ещё совсем нестарых устройств.

А что же в Linux? Если оборудование поддерживалось в предыдущих версиях таких распространённых дистрибутивов, как Ubuntu или Mint, то независимо от разрядности новых версий системы оно будет продолжать поддерживаться и в них после обновления.

С чем не согласна:

Практически вся информация, представленная в этой части книги является для меня новой, поэтому не согласиться мне вообще не с чем, а согласиться только чуть-чуть.

С чем согласна:

Согласна с пунктом о неотвратимости прогресса. Действительно, совсем не всегда обновления являются полезными, нужными или просто понравившимися.

С потерей части функциональности при обновлении я еще не сталкивалась, слава богу, пока только с удобством интерфейса. Например, выход Windows 8 дал мне сильный толчок для перехода на Linux. К слову, вообще не поняла, где распространено мнение о неподдерживаемости периферийных устройств на Linux. Только возгласы о том, как классно не качать драйвера.

Проектирование и процессы

Корпоративные информационные системы (КИС) прошли долгий путь от полной закрытости сплавленных с аппаратурой монолитов до создания модульных и открытых систем. Однако теперь вместо стандартизации процессов и реализации лучших практик создания базовой функциональности на передний план выходят конкурентные преимущества за счёт дифференциации и специализации. Прежде всего, за счёт обрастания «скелета» КИС «мышцами и кожей» специфичных для данного предприятия программ.

Слоистость и уровни

Любая автоматизированная информационная система может быть рассмотрена с трёх точек зрения проектировщика:

• концептуальное устройство;

• логическое устройство;

• физическое устройство.

Концептуальное устройство

Концептуальное устройство автоматизированных информационных систем составляют всего 3 слоя.

Слои концептуального устройства существуют в

любой информационной системе, даже если с точки зрения физической архитектуры или конкретного программиста их трудно различить. Это тот случай, когда незнание закона не освобождает от ответственности.

Логическое устройство

Напротив, слои логической архитектуры не являются строго определёнными. Основным способом их выделения является постоянный диалог проектировщика с требованиями к системе и ответами на вопрос «Зачем нужен этот слой в данном случае?». Наиболее типовые вопросы и ответы сведены в так называемые шаблонные решения и рекомендуемые практики.

Физическое устройство

Аналогично логической архитектуре, физическое устройство тоже является предметом синтеза на стадии проектирования реализации. Физический слой

также называется звеном (tier). Число звеньев системы определяется максимальным количеством процессов клиентсерверной архитектуры, составляющих цепочку между концептуальными слоями.

Тонким клиентом (thin client) традиционно называют приложение, реализующее исключительно логику отображения информации. В классическом варианте

это алфавитно-цифровой терминал, в более современном – веб-браузер.

В противоположность тонкому, толстый клиент (rich client) реализует прикладную логику обработки данных независимо от сервера. Это автономное приложение, использующее сервер в качестве хранилища данных. В трёхзвенной архитектуре толстым клиентом по отношению к СУБД является

сервер приложений. Так называемый «умный» (smart client) клиент по сути остаётся промежуточным решением между тонким и толстым собратьями. Будучи потенциально готовым к работе в режиме отсоединения от сервера, он кэширует данные, берет на себя необходимую часть обработки и максимально использует возможности операционной среды для отображения информации.

Многозвенная архитектура

Итак, концептуальных слоёв в автоматизированной информационной системе всегда три: хранения данных, их обработки и отображения. А вот физических, их реализующих, может быть от одного, в виде настольного приложения с индексированным файловым хранилищем, до теоретической бесконечности.

Имея опыт разработки систем всех перечисленных типов, наиболее позитивные впечатления у меня остались от «двухзвенки» с тонким клиентом.

Разумеется, у каждой архитектуры есть свои преимущества и недостатки. Если подходить к вопросу проектирования объективно, то выбор в каждом конкретном случае вполне рационален. Но я вспоминаю, например, что в начале профессиональной деятельности нам хотелось просто попробовать «сделать

трёхзвенку» своими руками, невзирая на то, что экономическая целесообразность такого выбора была не совсем очевидна как по затратам на разработку, так и по стоимости владения системой в будущем. Сегодня доля специалистов, имеющих опыт в системном проектировании, в общем потоке становится всё меньше. Поэтому декомпозиция и выбор архитектуры часто проводится по принципу «прочитали статью, у этих парней получилось, сделаем и мы так же». Огород из нескольких физических уровней насаждается не потому, что это действительно надо, а потому, что очередной гуру написал об этом в своём блоге. Или потому, что нет другого опыта и мотивации его приобрести: чему научили ремесленника, тому и быть.

В итоге между экраном конечных пользователей и запрашиваемыми ими данными образуется толстая прослойка, которая на 80 % занята совершенно пустой работой по перекачиванию исходной информации из одного формата представления в другой. Растёт время отклика системы. Пользователь нервничает и справедливо обвиняет в этом программистов.

Краткий отзыв?

Так получилось, что эта часть книги оказалась сложной для написания своего мнения. Очень много познавательной информации (описание уровней, например), которая просто принимается как данное. Но там закрались философские рассуждения на тему сколько звеньев оптимально для хорошей архитектуры. Я думаю, что все зависит от конкретного проекта, но пример про блоги гуру хорош и, на мой взгляд, справедлив, в бесконечность уходить точно не стоит.

Интервью, честно говоря, было не всегда понятным из-за упоминания систем, для которых я слишком молода, не сталкивалась, поэтому воспринимать сложно. Однако тут главное сама суть, правильно — совсем небольшая группка толковых людей делает за короткие сроки очень классную штуку. Потому что знают свое дело, а еще потратили 3 месяца только на построение архитектуры и разговоры по теме. Хотела бы я стать такой в будущем.

Тема эссе:

«Современные команды 10-15 человек, вооружённые умопомрачительными средствами рефакторинга и организованными процедурами гибкой разработки, за год не могли родить

работоспособный заказной проект, решающий несколько специфичных для предприятия задач.» Причины сложившейся ситуации.

Ultima-S – КИС из коробки

Существует два основных подхода к разработке КИС, условно называемых «от производства» и «от бухгалтерии». В первом случае функциональным ядром системы становится планирование ресурсов производства. Упрощенно: на предприятии есть сотрудники, оборудование и сырьё (материалы, компоненты), с одной стороны, а с другой – план выпуска – «чего и сколько?» вкупе с технологией – «как?», то есть правилами, нормами и прочими ограничениями процесса преобразования сырья в готовую продукцию. В первом приближении, необходимо составить оптимальный по загрузке персонала и оборудования план этого процесса. После того как система научилась составлять производственный план, она тянет за собой все остальные функции: от кадров, ведь персонал не из воздуха появляется, и снабжения сырьём до складирования и сбыта готовой продукции. Альтернативный путь проходит через бухгалтерию. Под термином «бухгалтерия» имеется в виду прежде всего внутренний, управленческий учёт на предприятии, а не фискальная её часть. Дело в том, что механизмы бухучёта придумали ещё в XV веке вовсе не ради подачи отчётности в средневековую налоговую инспекцию, а для понимания состояния дел на своём предприятии. Бухгалтерский учёт – хорошо формализуемая аппаратом матричной алгебры абстракция для отражения и анализа хозяйственных операций в дискретных периодах времени, в том числе и будущих.

Логическое устройство

Напротив, слои логической архитектуры не являются строго определёнными. Основным способом их выделения является постоянный диалог проектировщика с требованиями к системе и ответами на вопрос «Зачем нужен этот слой в данном случае?». Наиболее типовые вопросы и ответы сведены в так называемые шаблонные решения и рекомендуемые практики.

Проектированием системы на самом деле должен заниматься не технарь, а маркетолог как Стив Джобс. Система – это товар, удовлетворяющий потребности.

Маркетолог может понять потребности рынка, сформулировать требования к товару,

может оценить осуществимость маркетинговой компании. На деле технический архитектор нужен маркетологу для оценки себестоимости проекта и его сроков, а также для информации о новых перспективных технологиях, тогда маркетолог сможет искать сочетания «потребность + технология». Проблема проекта Ultima-S и многих других в том, что там не было маркетолога, определяющего вид продукта и программу продвижения, что сделало бессмысленным большое количество технологических действий.

большое количество технологических действий.

Следуйте совету Джобса: «Обычные художники заимствуют – великие воруют».

Прикладная разработка не место для эстетического наслаждения от красот технологий. Это бизнес.

Программисты и технологии могут выглядеть невзрачно, но давать невероятные эффекты по скорости и надёжности кода в терминах низких трудозатрат, и наоборот, вроде бы красивые решения могут превращаться в «чёрную дыру» для бюджета проекта.

Думать головой

Откажитесь от термина «наследование», который искажает смысл действий. Мы обобщаем. Обобщение (наследование) реализации или интерфейсов – весьма неоднозначный механизм в объектно-ориентированном программировании, его применение требует осторожности и обоснования.

Обобщаемые классы должны иметь сходную основную функциональность.

Например, если два или более класса:

• порождают объекты, реализующие близкие интерфейсы;

• занимаются различающейся обработкой одних и тех же входных данных;

• предоставляют единый интерфейс доступа к другим данным, операциям или к сервису.

Взгляните, три примера, и уже несколько шаблонов оказались ненужными: (1) – «фабричный метод» (factory method), (2) – «стратегия» (strategy), «шаблон метода» (template method) и (3) – «адаптер» (adapter). В принципе, можно для случая (1) ещё и «прототип» (prototype) записать: если в среде нет развитой поддержки метаинформации типа отражения (reflection), то реализовать клонирование, скорее всего, придётся в потомках общего предка. Далее, в большинстве случаев (если не во всех) вместо «посетитель» (visitor) можно использовать все тот же «шаблонный метод» или вызов метода через reflection. Ещё один исключаем. Почему шаблоны «вдруг» стали ненужными? Потому что требуется только обобщение, причём в перечисленных случаях необходимость операции более чем очевидна. Требования же исходят напрямую из вашей задачи. Корректно проведя обобщение вы автоматически получите код и структуру, близкую к той, что вам предлагают зазубрить и воспроизводить авторы разнообразных учебников шаблонов.

Глубина более двух уровней при моделировании объектов предметной области,

вероятнее всего, свидетельствует об ошибках проектирования.

Для построения устойчивой глобальной иерархии необходим серьёзный анализ предметной области, ведь не случайно создание таксономии – сложная научно-исследовательская работа, которой в крупных компаниях занимаются аналитики.

Про сборку мусора и агрегацию

Итак, сборщик мусора, он же GC – garbage collector в средах программирования с автоматическим управлением памятью. Наиболее очевидное преимущество – программисту не надо заботиться об освобождении памяти. Хотя при этом все равно нужно думать об освобождении других ресурсов, но сборщик опускает планку требуемой квалификации и тем самым повышает массовость использования среды. Но за все приходится платить. С практической стороны

недостатки сборщика известны, на эту тему сломано много копий и написано статей, поэтому останавливаться на них я не буду. В ряде случаев недостатки являются преимуществами, в других – наоборот. Черно-белых оценок здесь нет. В конце концов, выбор может лежать и в области психологии: например, я не люблю, когда компьютер пытается управлять, не оставляя разработчику достаточных средств влияния на ход процесса.

Нужно взять за правило, что контейнер всегда управляет своими объектами. Поэтому обращаться к его внутренним объектам нужно только через интерфейс самого контейнера. При этом быть готовым к обработке ситуации, когда контейнер говорит: «Извини, но такой объект уже удалён или пока недоступен». Если же объект переходит во владение к другому контейнеру, то он перестаёт управляться прежним. И процесс передачи объекта и управления также не может быть реализован простым присваиванием полученной ссылки.

«Проектированием системы на самом деле должен заниматься не технарь, а маркетолог». Отчасти я соглашусь с этим, особенно учитывая еще одну цитату: «Прикладная разработка не место для эстетического наслаждения от красот технологий. Это бизнес. ». У технаря слишком велики шансы

как раз это эстетическое наслаждение и получить. Продуктом будут пользоваться клиенты, на которых опираются маркетологи, вроде получается, что маркетолог на этапе проектирования важнее. Разработчиков (технарей) же меньше, да и вообще о том, что решение на самом-то деле красивое, будут знать только они. Лично я бы доверилась маркетологу при проектировании, а потом уже тому, что имею, постаралась добавить изящности в небольших моментах, не ломая основную концепцию.

Что касается паттернов. Любое шаблонное решение может проигрывать решению, найденному для конкретно поставленной задачи. Я думаю, это очевидно, не понимаю, почему автор данной книги так взъелся на «большую четверку». Да, я согласна с тем, что начинающему программисту, который хочет стать действительно профессионалом, лучше сначала самому подумать и понаступать на грабли, а только потом читать упомянутую книгу. Но мне кажется, сегодня разрабатывается столько всевозможных приложений, существует столько всевозможных проектов, что программистов, для которых главное не деньги, а сам процесс и самосовершенствование, немало. Но они же должны как-то решать возникающие проблемы. Книга по паттернам им в помощь, незаменима просто.

Сборщик мусора. Как и автора данной книги, меня настораживает, когда компьютер делает что-то за тебя, что ты не можешь контролировать. Например, в Java ты можешь только порекомендовать почистить память, но совсем ведь необязательно, что сборщик мусора тебя послушает. А еще мысль о том, что надо не забыть убрать за собой, держит в тонусе. Лично меня это даже заставляет задуматься, а нельзя ли использовать меньше памяти в своей программе. Когда же есть сборщик мусора, про память вообще почти не вспоминаешь. Хотя, может, это только у меня. В любом случае, вариант с самостоятельной уборкой мусора считаю очень полезным.

Журнал хозяйственных операций

- Неэффективно считать остаток по журналу

- Необходимы транзакции на уровне изоляции «сериализация», причем блокировать весь журнал нельзя

- Нужна таблица для поддержания остатков на заданный период в актуальном состоянии

UML и птолемеевские системы

Итак, 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.

Поэтому использование UML имеет две основные альтернативы, напрямую зависящие от целей:

• Цель – использовать «как есть». Не заниматься вопросами целостности, ограничившись рисованием частей системы в разных ракурсах. Если повезёт, то часть кода можно будет генерировать из схем.

• Цель – использовать для моделирования и генерации кода. Придётся создать свои формализмы, соответствующие моделируемой области. В самом минимальном варианте – использовать в принудительном порядке стереотипы и написанные руками скрипты и ограничения для проверки непротиворечивости такого использования. Чтобы, например, стереотип «Водитель» в рамках ассоциаций «Управление машинами» был связан только с классом, реализующим интерфейс

«Транспортное средство». Во втором случае формализация модели является, по сути, созданием предметно-ориентированного языка, то есть серьёзной исследовательской работой вплоть до уровня научной диссертации. Далеко не каждая софтостроительная фирма может позволить себе вести такие работы без ясных перспектив тиражирования своих продуктов.

Краткий отзыв

«Журнал хозяйственных операций». Что тут сказать, автор дал советы, как сделать так, чтоб система работало хорошо. Правда, мне кажется, он немного преуменьшает умственные способности начинающего программиста. Например, использование мьютекса в рассматриваемой системе более чем очевидно. А еще меня смущает тот факт, что то автор ругает паттерны за то, что миру представляются готовые решения, которые дают возможность программисту меньше думать, то сам преподносит на блюдечке это самое готовое решение.

«UML и птолемеевские системы». Вообще, я сама всякими утилитами для создания UML диаграмм не пользовалась. Но, конечно, на этапе проектирования не обходилась без листиков с рисуночками, описывающих сущности приложения и их взаимодействие. Мне их хватало, наверное, поэтому я не вижу смысла в проверке системой правильности диаграммы (отсутствие циклов и т. д.) и генерации кода. Но это потому, что я не сталкивалась с крупными проектами. Все-таки в больших масштабах первый пункт может быть очень полезен, сложно уследить за большим количеством компонентов и их связями. А к идее генерации кода я отношусь немного скептически. Мне кажется, нельзя на это полностью полагаться, нужно проверить, что ж там сгенерировалось. Но ведь велик соблазн оставить, как есть, правильно. Так что я бы делала ручками, да, много рутинного кода, но 90% это повторяющиеся блоки. Зато можно быть уверенной, что в ошибках виноват ты, а не система. Первый вариант менее обидный, как мне кажется.

Когда старая школа молода

«Оптисток», или распределённый анализ данных

Архитектура сокрытия проблем

Честно говоря, даже написать нечего. Я даже второй раз пробежала по

тексту в поисках чего-то нового. Автор в который раз указывает на то,

что, на самом деле, немного программистов, которые подходят к задаче

добросовестно и с умом. Довольно интересно читать эти истории, но,

увы, я не знаю, что в этой части стоит конспектировать.

Code revision, или Коза кричала

Ревизия кода, несомненно, весьма полезная процедура, но как минимум при двух условиях:

• эта процедура регулярная и запускается с момента написания самых первых тысяч строк;

• процедуру проводят специалисты, имеющие представление о системе в целом. Потому что отловить бесполезную цепочку условных переходов может и компилятор, а вот как отсутствие контекста транзакции в обработке повлияет на результат, определит только опытный программист.

Дж. Фокс выводит из своего опыта проектной работы в IBM важную мысль, что большой ошибкой является привлечение к процессу внутреннего тестирования и обеспечения качества посредственных программистов. По его мнению, компетентность специалиста в этом процессе должна быть не ниже архитектора соответствующей подсистемы. Действительно, ведь оба работают примерно на одном уровне, просто один занят анализом, а другой – синтезом.

Качество кода во многом зависит от степени повторного использования, поэтому приведу простой и доступный способ проверки того, не занимается ли

ваша команда программистов копированием готовых кусков вместо их факторизации. Для этого регулярно делайте сжатый архив исходников, например zip с обычным коэффициентом компрессии, и оценивайте динамику роста его размера относительно количества строк. Если размер архива растёт медленнее, чем количество строк, это означает рост размера кода за счёт его копирования.

«Наживулька» или гибкость?

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.

Если стадии, органично совпадающие с концептуальным, логическим и физическим дизайном системы, расположить иерархически без обратных связей,

то получим классическую схему «водопад», исторически считающуюся первой методологией в софтостроении.

Что же не так в схеме? С увеличением сложности реализуемой системы анализ и следующее за ним проектирование начинают занимать всё больше времени. Постепенно обнаруживаются новые детали и подробности, изменяются требования к системе, возникают новые сопутствующие задачи, расширяющие

периметр. Приходится, не отдавая проектную документацию в разработку, возвращаться к анализу. Возникает риск зацикливания процесса без конечного выхода какого-либо программного обеспечения вообще.

Но если проблема зацикливания на требованиях может быть успешно решена выбором подрядчиков, уже имевших опыт в построении систем данного типа,

то другая, гораздо более значимая проблема несовпадения взглядов заказчика и подрядчика на, казалось бы, одни и те же вещи стабильно проявляется с ростом

проекта. Даже если принять во внимание, что только 20 % требований специфичны для заказчика, тогда как 80 % исходят непосредственно от предметной области инвариантно среде и контексту, то эти 20 % способны угробить весь проект.

Для снижения рисков такого рода в конце 1980-х годов была предложена спиральная модель софтостроения.

Ключевой особенностью спиральной технологии является прототипирование. В конце каждого витка после этапа стабилизации заказчик получает в своё распоряжение ограниченно работающий прототип целой системы, а не отдельных функций. Основная цель прототипа состоит в максимально возможном сближении взглядов заказчика и подрядчика на систему в целом и выявлении противоречивых требований. Спиральная модель не навязывает присутствие всех стадий на каждом витке. Вполне может статься, что первый же прототип будет удачным, а функциональная и техническая архитектуры соответствуют требованиям. Тогда финальные витки будут фактически состоять только из разработки и стабилизации. Слабым звеном в спиральной методологии является определение длительности очередного витка,

его стадий и соответствующее выработанному плану управление ресурсами. Пока анализ не выдал концептуальные модели, проектировщики и ведущие

программисты ограничены техническими требованиями, тогда как рядовые программисты просто ожидают спецификации. Но неудачно сократив фазу анализа или проектирования на первом витке, тем самым можно увеличить их общее количество, рискуя выйти за рамки первоначальной оценки сроков и бюджета.

Ключевой особенностью гибкой методики является наличие мифологического титана – владельца продукта (product owner), который лучше всех знает, что

должно получиться в итоге. На самом деле это просто иная формулировка старого правила «кто платит, тот и заказывает музыку». Именно владелец, за рамками собственно гибкого процесса, гением своего разума проводит анализ и функциональное проектирование, подавая команде на вход уже готовые пачки требований. Размер пачки должен укладываться в интеллектуальные и технологические возможности разработчиков, которым предстоит осуществить её реализацию за одну итерацию. В итоге мы получаем знакомую софтостроительную схему «снизу-вверх», появившуюся на свет гораздо

раньше «водопада», с его упорядочивающей моделью «сверху-вниз». То есть мы не знаем точно, что хотим получить в целом, но знаем отдельные функции,

реализовав которые, мы, возможно, придём к решению.

Для программистов позитив, к сожалению, изрядно разбавлен издержками производственного процесса. Методология, по сути, направлена на увеличение времени работы с клавиатурой и не располагает к размышлениям. Пиши код! Поэтому для стимуляции персонала процесс окружен религиозной атрибутикой, манипуляциями, иносказаниями и метафорами. С другой стороны, требования к уровню программиста ограничиваются знанием конкретных технологий кодирования, стандартных фреймворков, «умением разбираться в чужом коде» и «умением работать в команде», уже упоминавшимся в словаре для начинающего соискателя. Способность решать олимпиадные задачки здесь от вас не требуется. Скорее, наоборот, будет помехой.

Позитив для заказчика в том, что, 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. После чего он может переформулировать существующие и добавлять новые требования с учётом уже набитых шишек. Тем не менее с ростом сложности системы возрастает и риск увеличения стоимости внесения изменений. И если проект выходит за рамки бюджета, то «козлом отпущения» становится именно владелец продукта.

Другой позитив для компании-заказчика состоит в непосредственной близости выполняемой подрядчиком работы. Зачастую «гибкие» команды работают на

площадке компании и доступны в любой рабочий момент. Если заинтересованному лицу не хватает информации, он может просто подойти и посмотреть на месте, поговорить с исполнителем и тем самым восстановить расстроившееся было душевное равновесие.

Позитив для подрядчика состоит в том, что «гибкая» разработка позволяет вовлечь в проект как можно больше разработчиков с менее высокими требованиями к квалификации. Это позволяет содержать больше сотрудников в штате, включая оффшорные команды. Будучи поставленным перед выбором между небольшой программистской фирмой с квалифицированным персоналом и софтверхаузом-«тысячником», крупный заказчик в общем случае склонится ко второму варианту.

Тенденция взаимного перекладывания ответственности на сложных проектах. Заказчик осознаёт, что реализовать спецификации своими силами невозможно,

прежде всего потому, что при таком объёме они тем не менее неполные и неизбежно содержат противоречия. Подрядчику же в принципе наплевать на спецификации, он будет крутить итерации, честно реализуя заявленный функционал и отрабатывая бюджет. Получился коровник на подпорках с покосившимися заборами и дырявой крышей вместо современного агрокомплекса? Извините, всё по спецификации, каждые две-три недели вы видели расцвеченные фотографии разных участков возводимого сооружения.

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

Тесты и практика продуктового софтостроения

Сложилось впечатление, что тотальные модульные тесты хорошо использовать для продуктов, которые реализуют какой-то стандарт или спецификацию (СУБД, веб-сервер), но для тиражируемого веб-приложения это смертельно.

Причины:

1. При изменении спецификаций затраты времени на приведение в актуальные состояния тестов могут быть куда больше, чем на, собственно, код.

2. Сбои, которые могут выловить тесты (повторяемый сбой в модуле, ранее уже исправлявшийся), – довольно редкое дело. Гораздо чаще сбои возникают в интеграции разных технологий, которые сложно автоматически протестировать.

3. Наличие модульных тестов сильно затрудняет масштабный рефакторинг. Если у нашего приложения есть какой-то внешний API, то все, что ниже, мы можем менять быстро и без особых проблем. Но если для этого низкоуровневого кода есть тесты, то их придётся основательно переделывать.

4. Если не напрягаться с выпуском раз в 2 недели, то возможность быстро что-то

протестировать не так уж и важна.

5. Частый выпуск версий просто не позволяет довести код до ума и провоцирует исправление старых и серьёзных проблем методом написания «залепени».

6. Я совсем не уверен, что пользователи хотят получать новую версию раз в 2 недели. Если они купили продукт – значит, он делает то, что они хотят. Если

чего-то не делает – они все равно уже нашли workaround (обходное решение), им всё равно.

7. Для корпоративного софта пользователи хотят длительного периода поддержки, пара лет, минимум. Если мы выпускаем продукт раз в 2 недели и находится серьёзная ошибка в версии годичной давности, то нам её нужно будет исправить примерно в 40 ветках. Конечно, править 40 веток кода никто не будет, пользователям сообщат, что ошибка будет исправлена в следующей версии, многие пользователи перейти на неё не смогут (см. п.6) к большой радости конкурентов с предложениями типа competitive upgrade offer.

Разработка модульных тестов – это тоже разработка. Для 100 % покрытия потребуется примерно столько же времени, сколько и на основную работу. А может, и больше, смотря как подойти к делу. По моим наблюдениям, соотношение объёмов рабочего и тестирующего кода примерно 1 к 2.

В отличие от тестов функциональных, завязанных на интерфейсы подсистем, модульные тесты требуют переработки одновременно с рефакторингом рабочего кода. Это увеличивает время на внесение изменений и ограничивает их масштаб, приучая разработчика минимизировать реструктуризацию, заменяя её надстройкой, быстро трансформирующей архитектуру в Ад Паттернов.

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.

Наконец, модульный тест – это самый нижний уровень проверок. Прохождение серии модульных тестов вовсе не гарантирует, что пройдут функциональные

тесты.

Говорящие изменения в MSF и выключатель

Интерес представляет следующий факт: в своей текущей редакции MSF 4.0 (Microsoft Solution Framework – методология софтостроения, опирающаяся на практики Microsoft.) была разделена на два направления: MSF for Agile Software Development и MSF for CMMI Process Improvement. CMMI, или Capability Maturity Model Integration, – модель зрелости процессов организации вообще и

софтостроения в частности. Ключевое слово здесь именно «зрелость», описываемая в CMMI несколькими уровнями организации: от хаоса до системы качества.

Соответственно, одно из нынешних направлений MSF предназначено для достижения зрелости процессов софтостроения или дальнейшего улучшения

процессов уже более-менее зрелых организаций. А второе... для гибкой разработки.

Качество программного продукта – многозначное и сложное понятие. Производственная культура – ещё более сложное. В одном можно быть уверенным: ни о какой культуре софтостроения не может идти и речи, если любой программист из коллектива не способен остановить бессмысленный циклический процесс для выяснения, какого же рожна по историям заказчика потребовалось обобщать четырёхногих коров и обеденные столы.

Краткий отзыв

Ревизия кода. Очень согласна с тем, что начинать ее делать надо чем раньше, тем лучше. Мне кажется, это прозрачно — потом не придется недоумевать, почему не работает функция, и искать ошибку в огромной куче написанного кода.

По поводу высказывания, что ревизию должен осуществлять опытный программист. Конечно, опытный. Но и проектировать систему должен опытный программист, и реализовывать функционал, и тестировать. Везде нужны опытные программисты, не думаю, что этот факт нужно упоминать отдельно.

По поводу качества кода и повторного его использования. Я думаю, что код, который прошел ревизию, можно использовать, почему бы и нет.

Все перечисленные методологии мы уже рассматривали на занятиях, по водопадной модели даже писали эссе, не считаю нужным много разглагольствовать на эту тему. Конечно же все зависит от ситуации, методологию следует подбирать под проект. И я не считаю, еще забегая вперед на пункт «Говорящие изменения в MSF и выключатель», что гибкая методология это гиблое дело, и что достижения зрелости процессов софтостроения с ней не добиться. Просто другая.

Тестирование. На мой взгляд, довольно познавательный пункт вышел, комментарий Максима Крамаренко очень понравился, структурированный и обоснованный. Раньше я даже как-то не задумывалась о тестировании как о чем-то конкретном. Да, тесты нужны, конечно, без них никуда — говорили мы все время. А когда начинаешь погружаться глубже, понимаешь, что да, возможно, модульные тесты не так хороши, как хотелось бы. Функциональные тесты сейчас для меня кажутся куда лучше решением.

Программна фабрика: дайте мне модель, и я сдвину Землю

В управляемой моделями разработке (УМР) и в программной фабрике в частности наиболее интересной возможностью является генерация кода, скомпилировав который, можно сразу получить работающее приложение или его компоненты. Мы проектируем и сразу получаем нечто работающее, пусть даже на уровне прототипа. Уточняя модели, мы на каждом шаге имеем возможность видеть изменения в системе. Проектирование становится живым процессом

без отрыва от разработки.

У программистов «от сохи» отношение к CASE, как правило, негативное на уровне «я не верю, что какие-то картинки генерируют код лучше написанного руками». В таком отношении есть своя сермяжная правда, действительно, экскаватор, в отличие от мужика с лопатой, может вырыть не всякую яму. Доказывать обратное – бесполезная потеря времени.

У более продвинутых программистов, имевших опыт написания и сопровождения тысяч строк однотипного рутинного кода, претензии становятся обоснованными и касаются, как правило, следующих сторон применения CASE-средств:

• Если ручное написание кода принять за максимальную гибкость, то CASE может навязывать каркас, стиль кодирования и шаблоны генерации частей программ, ограничивающие не столько полёт фантазии программиста, сколько возможность тонкой настройки генерируемого кода. Неважно, что такая настройка не требуется в большинстве случаев, но если её нет, то менять придётся непосредственно сгенерированный код.

• CASE работает только в условиях дисциплины, когда ручные изменения генерируемого кода исключены или автоматизированы (пост-обработка). Как только программист залезает руками в код каркаса, модель оказывается рассинхронизированной по отношению к исходным текстам программ и процесс разваливается.

В качестве решения перечисленных проблем появились так называемые двусторонние CASE-инструменты (two way tools), позволяющие редактировать

как модель, непосредственно видя изменения в коде, так и, наоборот, менять код с полу– или полностью автоматической синхронизацией модели. Зачастую, такой инструмент был интегрирован прямо в среду разработки.

Нетрудно заметить, что двунаправленный подход в CASE-инструментарии в большей степени является мощным средством автоматизации отдельных программистов, так как обладает рядом ограничений:

• как правило, инструмент привязан к языкам и платформам;

• технология не выходит за рамки разработки конкретных программ и подсистем. То есть слои системы и архитектура остаются за рамками процесса;

• коллективная работа над моделями одновременно с кодом практически невозможна: приходится делить модели на независимые части, например подсистемы, разрабатываемые одним программистом;

• для достижения нужного эффекта методика по-прежнему требует навыков моделирования как минимум на уровне диаграммы классов. В противном случае CASE оказывается лишь очередным инструментов рефакторинга.

Следующим шагом в развитии автоматизированных средств софтостроения явилась программная фабрика – синтез подходов управляемой моделями разработки и архитектуры, генерирующий не только отдельные компоненты системы, но целые слои в соответствии с выбранной архитектурой и платформами.

Лампа, полная джинов

Переходя к техническим терминам, программист описывает задачу в терминах логической модели, представляющей собой набор сущностей, их атрибутов, операций и связей между ними. Язык создан на основе XML, поэтому делать описания можно непосредственно руками в обычном текстовом редакторе.

Модель в виде XML-файлов поступает на вход «заклинателю» – входящей в состав пакета консольной утилите. 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, касающуюся не только самих джиннов, но и, например, таких настроек, как правила именования в конкретном слое системы.

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. Конечно, даже теоретически такое восстановление не может быть полным из-за разницы в семантике, но большую часть рутинной работы оно выполняет. Проведя

один раз импорт, далее мы редактируем, структурируем модели и продолжаем работать только в обычном цикле изменений «через модель».

Остановиться и оглянуться

Если рассмотреть метрики относительно небольшого проекта, то 40 прикладных сущностей в модели, состоящей примерно из 600 строк XML-описаний, порождают:

• около 3 тысяч строк SQL-скриптов для каждой из целевых СУБД;

• порядка 10 тысяч строк домена;

• 1200 строк XML для проекций классов на реляционные структуры (таблицы);

• около 17 тысяч строк веб-служб и интерфейсов.

Таким образом, соотношение числа строк мета-кода описания модели к коду его реализации на конкретных архитектурах и платформах составляет около 600 к 30 тысячам или 1 к 50.

Это означает, что оснащённый средствами автоматизации программист с навыками моделирования на этапе разработки рутинного и специфичного для платформ/архитектур кода производителен примерно так же, как и его 50 коллег, не владеющих технологией генерации кода по моделям. Любое внесение изменений в модель тут же приводит в соответствие все генерируемые слои системы, что ещё более увеличивает разрыв по сравнению с ручными модификациями. Наконец, для генерируемого кода не нужны тесты. Производительность возрастает ещё как минимум вдвое.

Краткий отзыв

В данной части главы было представлено много практического руководства по использованию Genie Lamp, поэтому я это пропустила, оставив саму суть. Не знаю, честно говоря, почему это так подробно было расписано, мне кажется, это не та книга, которую будешь читать, чтобы научиться работать с Genie Lamp. Для того, чтобы показать, что это все делается несложно, мне кажется, пункт можно было сократить. Но ладно, это авторское право :)

Тема генерации кода, насколько я помню, уже подымалась и я уже писала свое мнение на этот счет. Наверное, это действительно помогает, когда у вас серьезная команда, занимаетесь серьезным большим проектом. Но вот лично мне, как начинающему разработчику, нравится больше вот такой вариант: просто возьмите и спихните эту грязную работу на джуниоров. Серьезно, вообще не смущает то, что это звучит как эксплуатация. Мне кажется, любой программист все-таки должен уметь написать то, что сгенерируется по модели. Для джуниоров это отличная возможность разобраться.

Cherchez le bug, или Программирование по-французски

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». Это ещё далеко не полный хаос, но уже давно и далеко не порядок. Все попытки как-то

организовать процесс разработки программ, всяческие методологии, применение парадигмы конвейера, стандарты и административные меры худо-бедно, но помогают оттянуть этот

критический порог на некоторое время. В идеале – до того момента, когда развитие системы останавливается и она, побыв некоторое

время в стабильном состоянии, потихоньку умирает. Одна из проблем организации промышленного производства программного обеспечения состоит в отсутствии каких-либо формальных описаний деятельности программиста. Можно определить в технологической карте, как работает сварщик или каменщик, но как пишет программу программист зачастую не знает и он сам. До художника, конечно, далеко не все дотягивают, а вот с деятельностью рядового журналиста «независимой» газеты непосвящённому в софтостроение человеку сравнивать вполне можно. Этакий ядрёный сплав ремесла, некого богемного искусства, со вкраплениями науки, вперемешку с халтурой, шабашкой и постоянным авралом. Попытки же принудить программиста делать однотипные операции противоречат самой цели существования программного обеспечения как самого гибкого из существующих средств автоматизации рутинных процессов и потому изначально обречены на неудачу.

Определить, что критический порог пройден, несложно, для этого у меня есть один простой признак: «Ты изменил чуть-чуть программу в одном месте, но вдруг появилась ошибка в другом, причём даже автор этого самого места не может сразу понять, в чем же дело». Существует и второй признак, не менее практичный: «Смотришь на чужой текст программы, и тебе не вполне понятен

смысл одной его половины, а вторую половину тебе хочется тут же полностью переписать».

При самых удачных раскладах борьба с багами может длиться годами. Например, в таком богатом и «багатом» учреждении, как Microsoft, несколько лет боролись за жизнь Windows 95, выпустили даже Windows 98, но в результате все-таки сил осталось только на Windows 2000. Для неспециалистов это может быть неочевидно, поэтому поверьте мне на слово, что эти системы совершенно разные, как дельфин и русалка. Второй способ: 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. Третий способ – выдать желаемое за действительное, побольше маркетинга, шуму, «подцепить» несколько заказчиков и на их деньги попытаться всё-таки перейти ко второму или первому способу.

Дефрагментация мозгов

Нынешняя «гуглизация» позволяет быстро находить недостающие фрагментарные знания, зачастую забываемые уже на следующий день, если не час. Поэтому ценность книг как систематизированного источника информации, казалось бы, должна только возрастать. Действительно, автору бессмысленно брать на себя функции интеллектуального фильтра RSS 130, составляя компиляции из содержимого чужих блогов. Кроме стоящих вне конкуренции учебников остаётся в основном только конкретизация – узкоспециализированная проработанная технологическая тема либо обобщение концептуальных наработок и практического опыта.

Много копий сломано в обывательских дискуссиях о так называемом клиповом мышлении, шаблонности, «плохом образовании» вкупе с апокалиптическими прогнозами и прочим. Хотя на самом деле пока непонятно, к чему приведёт тенденция в перспективе ближайших десятилетий. Во фрагментарном «клиповом» мышлении есть и свои плюсы: способность быстро решать типовые задачи, широкая квалификация и мобильность, меньшие затраты на массовое образование. Минусы тоже ясны. В апокалипсис технократии я не верю, всегда будут рождаться способные дети и существовать ограниченное число учебных заведений, дающих жутко дорогое фундаментальное образование, вроде того, что было общедоступным в СССР. Видимо, этого должно хватить на поддержку критичных технологий цивилизации и дальнейшее их развитие.

Простые правила чтения специальной литературы

Итак, настоящий исследователь должен:

• быть достаточно ленивым. Чтобы не делать лишнего, не ковыряться в мелочах;

• поменьше читать. Те, кто много читает, отвыкаютмсамостоятельно мыслить;

• быть непоследовательным, чтобы, не упуская цели, интересоваться и замечать побочные эффекты.

Технические книжки – это не бессмертные произведения графа Толстого. К ним должен быть суровый, сугубо индивидуальный и безжалостный подход без какой-либо оглядки на чужое мнение. Прежде чем начать читать, возьмите листок бумаги, оптимально формата А4 – сложенный пополам, он удачно вкладывается в книгу, служа одновременно и закладкой. И запаситесь карандашом. Пользователям устройств для чтения электронных книг, к коим я тоже отношусь, необходимо освоить функции вставки в текст своих заметок.

более функциональное.

Теперь, по мере чтения, начинайте делать пометки в стиле «с. 11, это мнение плохо обосновано, потому что...», «стр. 25, хорошая мысль, применимо также в

ситуации...». Очевидно, для пометок типа «у автора N выводы другие, но если обобщить...» потребуется немного больше времени. Если, пролистав с полсотни страниц, вы не сделаете ни одной пометки, то можете смело закрывать сие произведение. В топку бросать, наверное, не надо, но поберегите своё время. Если же к концу прочтения вам хватит листа А4, исписанного убористым почерком, для всех пометок, то можете считать, что чтение прошло не зря. Ну а если одного листа не хватило... Тогда смело идите на свой любимый интернет-форум и там поделитесь с коллегами своими находками. Когда вместо малоинформативного «must have», используя все тот же испещрённый пометками лист, вы сможете передать пользу от прочитанного, коллеги будут вам чрезвычайно признательны за рекомендации и собственное сэкономленное время.

Краткий отзыв

Эта часть, как ни странно, была самой интересной из всех в этой книге. Даже не очень хотелось ее конспектировать, нужно было просто записать страницы, которые надо будет перечитать еще раз на досуге.

Просто так получилось, что особенно близко ко мне оказались темы. У меня нет, конечно, как у автора этой книги, такого опыта работы. Да у меня вообще его пока почти нет. Но все-таки надеюсь, в собственном конспекте я могу позволить себе написать то, что хочу, историю из жизни, как это делает автор :)

По поводу критического порога проекта. Я почти уверена, что проект, над которым я сейчас работаю, его перешел. Когда я первый раз туда пришла, начала пытаться разбираться в коде, вникнуть во все это дело... Все очень плохо было. Ничего не понимала, что откуда берется, что происходит. Слава богу у меня очень хорошая (и очень небольшая, кстати, по крайней мере в нашем офисе) команда, мне помогали как могли. Так вот, объясняют мне что-то, я понимаю, дают мне задание, я ищу что-то похожее в проекте, так сказать, не нарушая общности. Нахожу, начинаю туда смотреть, а там что-то очень некрасивое. Я спрашиваю, а что это такое? А мне говорят, чтобы я туда не смотрела, это очень старый код, который вообще вряд ли работает. То есть получается, что сейчас проект — большая мусорка, в которой кто-то знает, что работает, а кто-то нет. Позже я узнала, что за основу этого проекта просто брали другой, который тоже уже очень-очень старый. И все это тянется уже

несколько лет, а релиз там вообще и не близко. Я знаю, что на этом проекте меня через год не будет, но что будут делать те, кто останутся, я просто не представляю.

Окей, теперь о фрагментарном мышлении. Фраза: «Нынешняя «гуглизация» позволяет быстро находить недостающие фрагментарные знания, зачастую забываемые уже на следующий день, если не час». Опять мини-история :)

Я слышала о том, что в промышленном программировании всем плевать, что ты не знаешь чего-то. Важно делать все так, чтобы это работало, в кратчайшие сроки. И тут понятно, что никак, кроме метода «гугления», не справиться. Никто не будет давать тебе время читать книги, документацию на рабочем месте. Для этого у тебя есть твое свободное время. Меня это очень удручает, потому что мне есть чем заняться в свое свободное время, и это не менее интересные вещи. Так как я не была уверена, где я вообще слышала это мнение, я решила спросить у своего куратора со стажировки. И о ужас! Он сказал то же самое. Это было как раз в то время, когда меня только взяли на работу. Я и так не очень-то уверена, что программирование - это то, что мне нужно.

А потом я попала на теперешний проект, о котором я уже рассказывала. И о команде я рассказывала. Что я хочу сказать: все зависит от людей, которые с тобой работают. Мой team-lead очень хороший в этом плане человек. Один раз он мне даже прямым текстом сказал: «Знаешь, я не знаю, как это будет работать. Почитай документацию, потом и мне заодно расскажешь». Все условия для обучения, так что не все потеряно!

Вот такой вот получился финальный конспект. В целом, мне эта книга не очень понравилась. Автор показался наглым и заносчивым :) Но тем не менее, там есть пункты, которые стоит будет перечитать в будущем. Я вообще думаю, что конкретно мне просто немного рано ее читать. В книге много моментов, над которыми надо подумать, но когда ты с этим не сталкивался, все это не так веско. Может быть, через 5 лет я вспомню об этой книге, будет интересно ☺

Трубач Геннадий

Кто такой программист и немного о резюме (стр. 11-57)

Конспект

Кто такой программист

Программист минус алгоритмизация = кодировщик

Программист минус кодирование = постановщик задачи

Практика многих десятилетий показала, что по-прежнему наиболее востребованным специалистом является инженер, способный как самостоятельно формализовать задачу, так и воспользоваться стандартными средствами её решения на ЭВМ. Вот его и следует называть *программистом*.

Иерархия подразделений

Компания
Дивизион (отделение)
Отдел
Группа
Группа

Иерархия должностей

«Б» – современные компании.

Уровень иерархии	Образовательный ценз (*)	А	Б	Англоязычный термин
1	Доктор (кандидат) наук	Директор	Директор (генеральный директор)	CEO — Chief Executive Officer
1	Доктор (кандидат) наук	Главный инженер	Технический директор	CTO — Chief Technology/Technical Officer
2	Доктор (кандидат) наук	Заместитель по отделению	Заместитель директора по направлению, директор по направлению	(**)
3	Кандидат наук	Начальник отдела	Начальник отдела	Head of service, head of department
4	Кандидат наук	Заведующий лабораторией	Руководитель группы	
5	Высшее образование	Заведующий сектором	Руководитель группы	Team leader
6	Высшее образование	Инженер (категории 3, 2 и 1, старший/ведущий), научный сотрудник (младший, старший)	Специалист, инженер	
7	Среднее специальное образование	Техник, лаборант, студент-практикант	Сотрудник	

Уровни принятия проектных решений

Уровень	Тип принимаемых решений
1	Целеполагание, технико-экономическое обоснование, регламент, бюджет, планирование
2	Требования к системе, концепция, техническое задание, архитектура системы

Функциональные специализации (роли)

«Б» – современные компании.

Уровень проектного решения	А	Б	Англоязычный термин
1	Руководитель проекта, научный руководитель	Руководитель проекта	Project Manager
2	ГИП (главный инженер проекта), главный конструктор (*)	Системный архитектор	System Architect
3	Ведущий инженер (область, специализация)	Архитектор подсистемы или направления (**)	Software architect, database architect, hardware architect
4	Инженер (область, специализация)	Разработчик (специализация)	Software engineer, database engineer, system engineer
5	Техник (область, специализация), оператор ЭВМ	Младший специалист, кодировщик	Software developer, database developer, web developer, support & helpdesk

Техническое творчество

Нельзя «сделать красиво», если относиться к работе исключительно утилитарно и шаблонно. Получаются сплошные типовые дома и «мыльные» сериалы. Необходимы и чувство прекрасного, и чувство меры, и знание других образцов, считающихся лучшими. Нужна техническая культура. Долгая работа, неблизкий путь, мотивация преодолеть который исходит, прежде всего, от любви к собственному делу, к профессии.

Но и нельзя «сделать красиво», если рассматривать софтостроение лишь как искусство и средство самовыражения. Любить себя в софтостроении, а не софтостроение в себе. Тогда красота рискует так и остаться не воплощёнными в жизнь эскизами. Невозможно обойтись без знаний технологий производства и хороших ремесленных навыков.

О квалификации

Отсюда неутешительный вывод для писавших программы в школьных кружках: количество проектов, где потребуется ваша квалификация, намного меньше количества некритичных заказов, а большинство ваших попыток проявить свои знания и умения столкнется с нелояльной конкуренцией со стороны вчерашних выпускников курсов профессиональной переориентации. На практике это означает, что вам, возможно, придётся снижать цену своего труда и готовиться к менее квалифицированной работе.

Не забывайте, что относительная доля критичных к качеству проектов падает, а переделка работающих систем базового уровня, от которых непосредственно зависит бизнес, – и вовсе редкое явление. Новые значимые проекты возникают только с новыми рынками и направлениями бизнеса.

Поэтому немало специалистов высокой квалификации уходят в экспертизу и консалтинг, где проводят аудит, обучение, «натаскивание» и эпизодически «вправляют мозги» разным группам разработчиков из числа переквалифицировавшихся.

Другой доступный вариант – специализация на предметных областях. В этом случае разработчик относительно автономен и, во-первых, гораздо менее ограничен в выборе инструментов. Во-вторых, что более существенно, доказывать кому-то степень владения инструментарием у него нет необходимости. К сожалению, хорошее знание предметных областей в сочетании с глубокими техническими знаниями платформ встречается редко в связи с плохой совместимостью высокоуровневых абстракций и низкоуровневых деталей. Обычная эволюция такого специалиста – системный аналитик, сохранивший знания технологий времён своего последнего сеанса кодирования в интегрированной среде.

Начинающим

Объявления вакансий:

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 компиляции 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.

Про CV (резюме)

1. **Краткость – сестра таланта**. Даже в небольшой фирме ваше резюме будут просматривать несколько человек. Вполне возможно, что первым фильтром будет ассистент по кадрам, который не имеет технического образования и вообще с трудом окончил среднюю школу. Поэтому постарайтесь на первой странице поместить *всю* основную информацию: ФИО, координаты, возраст, семейное положение, мобильность, личный сайт или блог, описания

своего профиля, цель соискания, основные технологии с оценкой степени владения (от «применял» до «эксперт»), образование, в том числе дополнительное, владение иностранными языками. Всё остальное поместите на 2–3 страницах.

2. **Кто ясно мыслит**, **тот ясно излагает**. Все формулировки должны быть ёмкими и краткими. Не пишите «узнавал у заказчика особенности некоторых бизнес-процессов в компании» или «разработал утилиту конвертации базы данных из старого в новый формат». Пишите «занимался постановкой задачи» или «обеспечил перенос данных в новую систему».
3. **Не фантазируйте**. Проверьте резюме на смысловые нестыковки. Если на первом листе значится «эксперт по C++», но при этом в опыте работы за последние 5 лет эта аббревиатура встречается один раз в трёх описаниях проектов, то необходимо скорректировать информацию.
4. **Тем более не врите**. Вряд ли кадровики будут звонить вашим предыдущим работодателям, но софтостроительный мир тесен, а чем выше квалификация и оплата труда, тем он теснее. Одного прокола будет достаточно для попадания в «чёрный список» компании, а затем через общение кадровиков и агентств по найму – ещё дальше.
5. Если соискание касается технического профиля, в каждом описании опыта работы **упирайте на технологии**, если управленческого – на периметр ответственности, если аналитического – на разнообразие опыта и широту кругозора.
6. **Не делайте ошибок**. Пользуйтесь хотя бы автоматической проверкой грамматики. Мало того, что ошибки производят негативное впечатление, они могут радикально изменить смысл фразы.
7. **Будьте готовы**, **что далее первой страницы ваше резюме читать не станут**, а о подробностях «творческого пути» попросят рассказать на первом собеседовании.

С чем согласен

Шуточная манера написания книги. Много полезных советов (например, для резюме). Сравнение того как было раньше в сфере IT, и как это сейчас. Разбиение на функциональные роли.

С чем не согласен

Опять-таки шуточная манера. Описание вакансий преподнесено так, как будто автора когда-то обидели. Очень много ненужного текста, который не имеет никого отношения к теме.

Железо и аппаратура (стр. 58-75)

Конспект

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.

Можно оценить количество состояний, которые необходимо охватить для полноты модульного тестирования при конвертации валют: ISO 4217 даёт список из 164 валют. Предположим, что наши входные данные:

- имеют только два знака после запятой;
- значения положительные;
- максимальная величина – 1 миллион;
- дата конвертации всегда текущая;

- мы используем только 10 валют из 164.

Итого: $10^2 \times 100\,000\,000 = 10\,000\,000\,000$.

На практике же программист применит эвристику и будет тестировать, например, только несколько значений (один миллион, ноль, случайная

величина из диапазона) для нескольких типовых конвертаций из 100 возможных, дополнительно проверяя допустимую точность значений на входе. При этом формальный показатель покрытия модульными тестами по-прежнему будет 100 %.

Также необходимо программировать тест производительности, который тем не менее *не гарантирует* время отклика, а только позволяет определить его *ожидаемое значение* при некоторых условиях.

Таким образом, собрав из кучи микросхем устройство, мы уверены, что оно будет работать:

- согласно таблицам истинности;
- с заданной тактовой частотой.

С чем согласен

Производительность «железа» возросла на порядки. Стоимость тоже на порядки, но снизилась. Увеличилась надёжность, развилась инфраструктура, особенно сетевая. Параллелизация вычислений пошла в массы на плечах многоядерных процессоров.

С чем не согласен

С тем, что ООП сыграла не последнюю роль в вытеснении женского труда из отрасли IT (стр 73). По моему мнению объектно-ориентированный подход ну никак не мог этому поспособствовать.

Также не согласен с тем, что «ООП реализовано повсеместно без малейших представлений о его применимости». ООП активно используется в нынешнее время. ООП принесло в программы такой важный момент, как безопасность, возможность наследования, полиморфизма.

Web-приложения (стр. 76-95)

С чем согласен

Согласен с написанным про апплеты. Они действительно были неудачной технологией. И также с написанным про создание web-приложений в прошлом, когда все писалось под конкретный браузер и что было непригодно после обновления браузера.

С чем не согласен

Так как материал совсем не для дискуссии, а больше повествовательно-описательный, здесь трудно выделить что-то, с чем резко не согласен. Соответственно тяжело и выделить какие-нибудь важные темы для заметки.

Паттерны, ООП и БД (стр. 96-130)

Конспект

Ад паттернов

Автор книги очень хорошо описывает проблему ООП в виде огромного количества паттернов, которые часто используются не там, где нужно. Вот какие проблемы выделяет автор:

- 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 сложных и многоуровневых конструкций в стиле «новый адаптер вызывает старый» по мере так называемого эволюционного развития системы;
- лоскутная интеграция существующих систем и создание поверх них новых слоёв API.

С чем согласен

В принципе, все сказанное про ORM, по моему мнению, верно. ORM действительно упрощает жизнь разработчикам, но одновременно с этим, программисты не хотят разбираться в БД и языке SQL. Всегда нужно по правильному, сначала разобраться с БД, потом как с ней работать и что такое SQL, а только потом переходить к ORM, так как на деле будет допускаться огромное количество ошибок, некоторые из которых не вылезут сразу.

С чем не согласен

Автор гнет очень жесткую линию относительно ООП. Он очень часто упоминает проблемы ООП. Но не нужно забывать, что каждая парадигма имеет свои плюсы и минусы. ООП является самым популярным подходом в настоящее время.

Также непонятна фраза о сложности распутывания кода в ООП. Мое личное мнение – это то, что разбирать чужой код в ООП намного проще, так как при ООП код довольно структурирован.

Агрегация и интерфейсы так же являются полезными вещами. Да, это все немного усложняет и увеличивает код. Но зато соблюдается структура и логика приложения.

Сервера и первая критика автора продукции Microsoft (стр. 130-169)

Коспект

Определение с Wikipedia

Мейнфре́йм (также мэйнфрейм, от англ. mainframe) — большой универсальный высокопроизводительный отказоустойчивый сервер со значительными ресурсами ввода-вывода, большим объёмом оперативной и внешней памяти, предназначенный для использования в критически важных системах (англ. mission-critical) с интенсивной пакетной и оперативной транзакционной обработкой.

Основной разработчик мейнфреймов — корпорация IBM, самые известные мейнфреймы были ею выпущены в рамках продуктовых линеек System/360, 370, 390, zSeries. В разное время мейнфреймы производили Hitachi, Bull, Unisys, DEC, Honeywell, Burroughs, Siemens, Amdahl, Fujitsu, в странах СЭВ выпускались мейнфреймы ЕС ЭВМ.

С чем согласен

«Как уберечь кукурузу от насекомых-вредителей? Очень просто: выкосить её всю, к чертям. Вредители придут, а кушать нечего.»
Так же согласен, что большие ЭВМ (типа мейнфреймов) являются неотъемлемой частью сервера. Никакие ПК не смогут заменить мейнфреймы!

С чем не согласен

И снова критика, еще с первых страниц. Очевидно, что Windows 7 в 1000000 раз лучше, чем Windows NT. Увеличение требуемой ОЗУ – это техническое требование, так возможности ОС возросли в огромное число раз.

И снова критика x2. Чем автору не угодил Office 2007? В 7-м офисе был значительно расширен функционал. Автор явно любит использовать старые софт и технологии.

Автор обиженно в конце упомянул, что он перешел на Libre Office, в котором редактировалась книга. Так вот хочу сказать, что оформление книги ужасно. Много сухого текста, коды оформлены как обычный текст и т.д. Ну и хочется пожелать автору удачи с таким отношением к продуктам

Microsoft. Если ему настолько все не нравится, то пусть переходит на ОС Linux (Unix, OS X, etc.) и пользуется самым “лучшим” и самым “удобным” софтом.

И как я и предположил, автор на 169 странице не забыл похвалить Linux, до этого сильно покритиковав продукцию Microsoft =) Не буду больше ничего писать, автор имеет право выражать свое предвзятое мнение.

Автоматизированная информационная система (стр. 170-201)

С чем согласен

Очень хорошо расписано про слои АИС. Со всем, написанным про нее, согласен.

Согласен с написанным в конце про то, что бывает такое: много человек не могут придумать как работать с проектом, а маленькая команда 4-5 человек может как раз и придумать.

С чем не согласен

Пункт про #ifdef. Зачем это вообще?

Управление предприятием, проектирование и сборка мусора (стр. 202-241)

Конспект

В первой части очень подробно описывалась разработка продукта управления предприятием. Описание получилось довольно интересным.
Сборщик мусора значительно упрощает процесс разработки. Но все же в нем есть не только плюсы, но и минусы. Например, плохо, если никому ненужный объект будет висеть в памяти. Но зато он уменьшает сложность языка.

С чем согласен

Проектированием системы должны заниматься не только технари. Все же должны участвовать люди, имеющие непосредственное отношение к сфере продукта (например, если это экономическая сфера, то проектировать должен заниматься человек, знающий ситуацию на рынке и в своей сфере)

Согласен с критикой книги Александреску «Мыслить шаблонно». Судя по написанному, там действительно много чего непонятного. Поэтому комментарии Тарасова очень даже удачны:

«Не увлекайтесь обобщением. Ошибки тоже обобщаются и уже в прямом смысле этого слова наследуются. Исправление по новому требованию может привести к необходимости сноса старой иерархии, содержащей ошибки.»

«Контейнер всегда управляет своими объектами. Поэтому обращаться к его внутренним объектам нужно только через интерфейс самого контейнера.»

С чем не согласен

Очень даже странно, но со всем написанным в этот раз соглашусь. Надеюсь, дальше темы будут интереснее

Транзакции и UML (стр. 242-259)

С чем согласен

Со всем согласен. Автор подробно и интересно расписал про важные вещи, такие как синхронизация транзакций. Это действительно нужная вещь, т.к. покупатель будет недоволен, если ему продадут не так, как он заказывал, что негативно скажется на репутации магазина.

UML диаграммы очень часто помогают разобраться что и как должно реализовываться. К тому же сейчас есть много средств их построения.

Распределенный анализ данных и Оптисток (стр. 260-283)

С чем согласен

Архитектура Оптисток выглядит довольно-таки интересно. Так как система является системой автоматизации продаж на выезде, то она является нужным и интересным продуктом. Как сказал сам автор, то архитектура типовая, но в его случае не обошлось без ограничений, а именно огромного числа пользователей. Но вместе с тем проблема была решена с помощью синхронизации и СУБД. Так же важен тот факт, что было уделено внимание GUI, так как интерфейс должен быть максимально понятным пользователю, чтобы упростить его работу.

С чем не согласен

К сожалению, для меня темы, описанные на данных страницах, совсем не вызывают интереса. И мое мнение, что здесь не о чем рассуждать, так как это скорее истории из жизни автора.

Ревизия кода, модели жизненного цикла и тестирование (стр. 284-310)

Конспект

Ревизия кода, несомненно, весьма полезная процедура, но как минимум при двух условиях:

- эта процедура регулярная и запускается с момента написания самых первых тысяч строк;
- процедуру проводят специалисты, имеющие представление о системе в целом. Потому что отловить бесполезную цепочку условных переходов может и компилятор, а вот как отсутствие контекста транзакции в обработке повлияет на результат, определит только опытный программист.

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.

Ключевой особенностью гибкой методики является наличие владельца продукта (*product owner*), который лучше всех знает, что должно получиться в итоге.

Модульного тестирование, является краеугольным камнем всех гибких методик: если изначально неизвестно, что выстроится в итоге, дом или коровник, то подпорки у его стен должны быть в любом случае.

С чем согласен

С необходимостью ревизии кода. Ревизия кода нужно не для слежения за выполнением проекта, но и для обучения молодых специалистов. Когда более опытный специалист производит ревизию кода молодого и указывает на ошибки, то это позволяет молодому специалисту поднять свой навык написания кода и предостеречь от появления ошибок.

Так же очень хорошо расписаны водопадная, итеративная и гибкие модели ЖЦ.

Владелец продукта — это просто иная формулировка старого правила «кто платит, тот и заказывает музыку».

Разработка модульных тестов – это тоже разработка. Для 100 % покрытия потребуется примерно столько же времени, сколько и на основную работу. А может, и больше, смотря как подойти к делу.

Практические выводы: соизмеряйте затраты на создание и поддержу автоматизированных модульных тестов с бюджетом проекта и располагаемым временем. Тем более плохо фанатично

навязывать разработку «от тестов», умалчивая о названных особенностях и не учитывая другие возможности. Например, код, генерируемый по моделям, вообще не требует модульных тестов.

Автору понравился подход MSF. Это очень радует, так как в предыдущих главах он сильно критиковал практически всю продукцию Microsoft (за исключением старых продуктов).

Ключевой особенностью системы в Тойоте является принцип «дзидока» (*jidoka*), означающий самостоятельность людей в управлении автоматизированной производственной линией. Если рабочий видит нарушение качества продукции или хода процесса, он имеет право, повернув соответствующий рубильник, остановить всю линию до установления причин дефектов и их устранения.

Качество программного продукта – многозначное и сложное понятие!

С чем не согласен

—

Система управления версиями и самогенерация кода (стр. 311-363)

Конспект

Team Foundation Server (сокр. TFS) — продукт корпорации Microsoft, представляющий собой комплексное решение, объединяющее в себе систему управления версиями, сбор данных, построение отчётов, отслеживание статусов и изменений по проекту и предназначенное для совместной работы над проектами по разработке программного обеспечения.

Subversion (также известная как «SVN») — свободная централизованная система управления версиями, официально выпущенная в 2004 году. Цель проекта — заменить собой распространенную на тот момент систему Concurrent Versions System (CVS), которая ныне считается устаревшей. Subversion реализует все основные функции CVS и свободна от ряда недостатков последней.

Лампа, полная джиннов

Есть модель, описанная с помощью XML. Она проверяется, чтобы не было противоречивых фактов. Если модель корректна, «заклинатель» начинает призывать «джиннов» сделать свою работу, передавая каждому на вход кроме самой модели ещё и разнообразные параметры, конфигурацию, касающуюся не только самих джиннов, но и, например, таких настроек, как правила именования в конкретном слое системы. 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.

Слои:

- Слой хранения (СУБД) (по сути отвечает за конфигурацию БД)
- Слой домена (NHibernate) (этот слой реализует такую важную вещь, как маппинг (mapping) классов на структуры хранения
- Слой веб-служб и интерфейсов доступа (ServiceStack) (формирует интерфейс для работы (различные классы)

С чем согласен

Довольно-таки поучительной получилась статья про TFS. По сути расписан один из возможных вариантов работы. Кратко его можно описать так: эту технологию можно, а ту нельзя; так, почему-то эта штука не настраивается, поищу-ка я в Google; О, вроде нашел, но оно все равно не

работает:(;…здесь такое понятие, как «танцы с бубном», т.е. попытки хоть как-то заставить работать нужную вещь; ну и завершается облегченным – «ну наконец-то!» Вот такая поучительная история:)

Также довольно интересно показалась идея про модель «Лампа, полная джиннов». Интересна тем, что она проста. И к тому же часто возникают мысли про что-то, что будет само генерировать код.

В принципе, дальше в книге описываются слои данной модели. Они являются вполне логичными, поэтому не имею ничего против. (слои написаны выше)

Эта модель так же завлекает тем, что, как утверждает автор, соотношение числа строк мета-кода описания модели к коду его реализации на конкретных архитектурах и платформах составляет около 1 к 50, что не может не радовать, так как писать нужно намного меньше кода.

С чем не согласен

—

Работа с непрофессионалами, утечка памяти, IBM, размышления о софтостроении и литературе и подведение итогов (стр. 364-404)

Конспект

Первые страниц 20 занимают рассказы из жизни автора. Он рассказывает, как "весело" он провел время в Париже и побывал целых три дня в IBM. Автор описывает, как он боролся там с насущными проблемами, а именно багами в коде, неопытностью или непрофессиональностью программистов и поджимающими сроками. Не буду вдаваться в подробности, но автор снова не забыл вспомнить о MS Windows (снова отрицательно; как сам сказал Тарасов: «в таком богатом и «**б а г** атом» учреждении, как Microsoft»).

Довольно мило автор описал основателей компании (хотя зачем это?), но все же описание приведу здесь: «Основателей у фирмы двое: генеральная директриса Софи – экспрессивная француженка, **незамужняя**, уже в **летах**, и технический директор Блез – почти наш ровесник, но, по-видимому, реально взявшийся за свой первый проект». Так вот, автор описывает проблемы работы с двумя продуктами этой «конторы». В первом он столкнулся с неспособностью делать что-нибудь полезное тамошних программистов и с тем, что продукт съедал аж 1 ГБ в день. Поработав в воскресенье, он со своим **русским** товарищем смог исправить утечку памяти, но осталась проблема взаимной блокировки. Что с ней стало, дальше не указывается.

Работая над вторым проектом, автор успел побывать в IBM (ровестник Блез смог пробить путевку в известную нам компанию). Причиной было то, что у заказчика был сервер данной компании. Сначала его друг пытался справиться сам, как хороший знаток OS Linux, но потом на подмогу ему был отправлен автор. Опущу описание переездов, скажу лишь, что это очень долго и нудно.

Первые два дня автор и его друг не могли исправить проблему с каким-то модулем (он не запускался). Если кратко, то в первые два дня автор и его друг успели изрядно покопаться в разработках IBM (компиляторе, библиотеках и т.д.), покушать Snickers и изрядно наездиться. На третий день, же они сходу нашли проблему в коде модуля, и подключив некоторые библиотеки статически, они запустили наконец этот модуль.

И да, успешно сдали его Блезу, который отдал его довольному заказчику.

Последующие страницы заняты размышлением о науке в софтостроении, правильном подборе и чтении литературы и кратким изложением «Оснований». Здесь я везде целиком соглашусь с автором. Действительно, сейчас литература уже не так популярна. Все всё «гуглят». Очень показателен пример автора, где он рассказывает о том, как практиковал обучение СУБД. Старшее

поколение пыталось вникнуть в детали основательно, а большинство из младшего — почти не обращало внимания, а лабораторные делали методом «тыка», выпрашивая код, чтобы сдать лабу. К сожалению, это очень правдиво. По учебе это видно, хватает таких личностей (но не все, конечно).

Правила чтения литературы просто приведу здесь. Там и так все хорошо написано:

> Хороших и полезных книг в принципе не может быть много. Иначе вы посвятите все рабочее время чтению. Тут уместно вспомнить второе правило Аникеева [12]; кстати, все три его правила стоит привести целиком. Итак, настоящий исследователь должен:
>
> - быть достаточно ленивым. Чтобы не делать лишнего, не ковыряться в мелочах;
> - поменьше читать. Те, кто много читает, отвыкают самостоятельно мыслить;
> - быть непоследовательным, чтобы, не упуская цели, интересоваться и замечать побочные эффекты.

Также понравилось следующее замечание, хотя скажу честно, сам нечасто делаю пометки. Может неинтересно мне эти темы... Так вот само замечание:

> Если, пролистав с полсотни страниц, вы не сделаете ни одной пометки, то можете смело закрывать сие произведение. В топку бросать, наверное, не надо, но поберегите своё время. Если же к концу прочтения вам хватит листа А4, исписанного убористым почерком, для всех пометок, то можете считать, что чтение прошло не зря. Ну а если одного листа не хватило… Тогда смело идите на свой любимый интернет-форум и там поделитесь с коллегами своими находками.

Дальше автор сравнил литературу и софтостроение. А действительно, их организации очень похожи!

Литература		**Софтостроение**	
Формат	**Содержание**	**Формат**	**Содержание**
Рассказ	Эпизод из жизни общества (сообщества)	Программа	Решение частной задачи
Повесть	Связные эпизоды из жизни общества (сообщества)	Программный пакет	Решение нескольких задач, как правило, связанных
Роман	Жизнь общества (сообщества)	Программная система	Решение задач предметной области
Роман-эпопея	Жизнь народа	Программный комплекс	Решение задач нескольких связанных предметных областей

Заканчивает автор краткой повестью об том, как все появилось. Попытаюсь кратко изложить основные тезисы автора:

1) Сначала были слова, потом образовались последовательности => инструкции. Так появилось программирование и машинный код.
2) Появляются различные новые функции.
3) Ребята помоложе, придумали использовать вместо машинного кода мнемокоды. Так зародились трансляторы, так как ЭВМ не понимает мнемокодов.
4) И снова новые функции.

5) Тут пришла мысль о том, а почему не писать программы на человеческом языке? Так зародился транслятор, который переводил человеческий язык в машинный код — Фортран (Формуло-Транслятор).
6) Появился КОБОЛ.
7) Пришла идея об типах переменных по умолчанию. И все переменные, начинающиеся на *i, j, k, l, m* и *n* Транслятор считал целочисленными.
8) Первые трансляторы требовали сложных инструкций. В целях упрощения появились Паскаль и С.
9) Ну а дальше пошли какие-то непонятные мне изменения: 4,5 система «Кофе» и т.д.

Закончу последним абзацем книги:

> И тогда все хором – от Старых Чудаков Первой Системы до Самых Молодых Парней – сказали: в нашей отрасли кризис! А Старые Чудаки добавляют: застой, и света в конце тоннеля не видно. Рядом с ними Маргинальные Чудаки с искусственным интеллектом под мышкой, специализированными ЯВУ и ЭВМ, суперкомпьютерами, векторными вычислениями и прочей экзотикой стоят в сторонке от мейнстрима и посмеиваются над ловкостью, с которой им удалось обмануть всех, включая самих себя.

С чем согласен

Не буду повторять выше описанное. Скажу в целом по книге. Я не сомневаюсь, автор — очень опытный и много чего повидавший человек. Он рассказывал обо многих насущных вещах в софтостроении и реальных попытках их исправить. Где-то удачно, где-то через «костыль», где-то неудачно, но главное, что попытки были. Так, конечно, книга была полезной. Но была бы еще полезнее, если бы в ней не было большого количества воды. Хотя это мое чисто субъективное мнение.

С чем не согласен

Сколько можно критиковать Microsoft? Да, они не без изъянов. Но никто не идеален и ничто не идеально!!

Критика Microsoft мне очень не понравилась. Я помню, что автор верстал книгу в LibreOffice — бесплатный офис. Но верстка — ужасна. Хотя может в печатной версии все довольно-таки неплохо.

Так же не понравилось высказывание о том, что из-за ООП из программирования ушли женщины. Может автор что-то другое имел ввиду, но я понимаю так, как есть.

Щавровский Святослав

Много времени прошло с момента появления первых ЭВМ. Поменялась структура, изменились технологии, увеличилась производительность, прошла революция миниатюризации. Не изменилась лишь суть программиста: человек, способный заставить компьютер сделать что-то.

За это время задачи, которые ставились перед программистами, крайне сильно прибавили в количестве. Это повлекло за собой появление новых сфер, которые спровоцировали появление новых специальностей. А как было раньше? Раньше было глобальное разделение на инженеров и техников. Формальное различие состояло в разной подготовке: техников готовили профессионально-технические училища и техникумы. Инженеров же готовили технические ВУЗы. Фактическое различие заключалось в различной сфере деятельности: техники занимались программированием и эксплуатацией, не влезая в постановку задач и проектирование. Современная ситуация несколько иная. Сейчас нередко диплом ВУЗа подтверждает лишь уровень техника. В Европе же, напротив, выпускники инженерных школ и университетов четко разделяются на техников и исследователей для научной работы соответственно.

Заведем разговор о красоте. Существует противоречивый термин - «техническое творчество». Раньше программирование можно было смело назвать ремеслом. Рабочим инструментом технического прогресса. Теперь же программирование нельзя причислить целиком ни к искусству, ни к ремеслу, ни к науке. Это случилось из-за катастрофически сильного развития сферы услуг, которая залезла в технологическую сферу. Пока одни программисты корпят над программами, другие реализуют свои интересы, вполне возможно, творческие, но не технические. Хорошо это или плохо - сказать трудно. Но одно точно: программирование изменилось. Как и изменилось отношение к нему. Уже не необходимо быть высококвалифицированным специалистом с большим багажом знаний, чтобы заставлять компьютер делать что-то. Нет необходимости перетруждать себя в доскональном изучении сектора своей компетенции. Представьте, что вне зависимости от того, будете вы работать или нет, государство будет выплачивать вам сумму, достаточную для нормального существования. Останетесь ли вы профессионалом в своей сфере? В не лукавом ответе на этот вопрос кроется проблема, которая вовсе не украшает it-сферу.

Софтостроение представляет собой соединение непосредственно продуктовой части (средства разработки, прикладные системы, пакеты) и рынка услуг. Причем отношение первых ко второму крайне мало. Согласно сведениям IBM, сообщество Java-разработчиков уже к 2006 году насчитывало более 6 миллионов человек. Нетрудно понять, что, чтобы обеспечить себе уровень жизни «выше среднего», вы должны держаться подальше от этих 6 миллионов. Необходимо понять: «количество проектов, где потребуется ваша квалификация, намного меньше количества некритичных заказов, а большинство ваших попыток проявить свои знания и умения столкнется с нелояльной конкуренцией со стороны вчерашних выпускников курсов профессиональной переориентации». Поэтому многие специалисты высокой квалификации уходят в экспертизу и консалтинг. Также вариантом является специализация в предметных областях. В постиндустриальной экономике сфера услуг занимает более 50 % деятельности, и эта доля растёт, например, в США она уже близка к 70 %. Для начинающих я составил небольшой словарик ключевых фраз, часто присутствующих в объявлении о вакансии. По замыслу, он должен помочь молодому соискателю вакансии программиста разобраться в ситуации и принять решение на основе более полной информации:

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».

Касаемо резюме. Вот несколько основополагающих принципов:

Краткость – сестра таланта. Даже в небольшой фирме ваше резюме будут просматривать несколько человек. Вполне возможно, что первым фильтром будет ассистент по кадрам, который не имеет технического образования и вообще с трудом окончил среднюю школу. Поэтому постарайтесь на первой странице поместить всю основную информацию: ФИО, координаты, возраст, семейное положение, мобильность, личный сайт или блог, описания своего профиля, цель соискания, основные технологии с оценкой степени владения (от «применял» до «эксперт»), образование, в том числе дополнительное, владение иностранными языками. Всё остальное поместите на 2–3 страницах.

• Кто ясно мыслит, тот ясно излагает. Все формулировки должны быть ёмкими и краткими. Не пишите «узнавал у заказчика особенности некоторых бизнес-процессов в компании» или «разработал утилиту конвертации базы данных из старого в новый формат». Пишите «занимался постановкой задачи» или «обеспечил перенос данных в новую систему».

• Не фантазируйте. Проверьте резюме на смысловые нестыковки. Если на первом листе значится «эксперт по C++», но при этом в опыте работы за последние 5 лет эта аббревиатура встречается один раз в трёх описаниях проектов, то необходимо скорректировать информацию.

• Тем более не врите. Вряд ли кадровики будут звонить вашим предыдущим работодателям, но софтостроительный мир тесен, а чем выше квалификация и оплата труда, тем он теснее. Одного прокола будет достаточно для попадания в «чёрный список» компании, а затем через общение кадровиков и агентств по найму – ещё дальше.

• Если соискание касается технического профиля, в каждом описании опыта работы упирайте на технологии, если управленческого – на периметр ответственности, если аналитического – на разнообразие опыта и широту кругозора.

• Не делайте ошибок. Пользуйтесь хотя бы автоматической проверкой грамматики. Мало того, что

ошибки производят негативное впечатление, они могут радикально изменить смысл фразы. Например, если написать «политехнический университет».

• Будьте готовы, что далее первой страницы ваше резюме читать не станут, а о подробностях «творческого пути» попросят рассказать на первом собеседовании.

С приходом глобальных поисковых сервисов, ценность информации снизилась. Притом существенно. Теперь ценно не столько владение информацией, сколько возможность получить её за ограниченный срок времени. Хорошо это или плохо - сложный вопрос. В связи с такой доступностью информации, технологии получили новый толчок. Нельзя сказать, что толчок в нужную сторону. Возможно случилось небольшое отклонение или виляние, но определенно толчок случился. Но осталось неизменным то, что технологии - основа нашей индустрии.

С увеличением количества сервисов, функциональности которых слегка (или не слегка) пересекаются, появилось совершенно понятное желание программистов сократить себе работу, а именно разработать новый «модульный» подход к разработке ПО. Модульность уже давно и легко реализуется в аппаратном конструировании. Это обусловлено жесткими входными и выходными данными последнего. В программном конструировании такого «совершенства» добиться сложно. Однако попытки были предприняты и предпринимаются до сих пор, притом с успехом. Хотя насчет «успеха» можно поспорить. Давно известно, что нет таблетки от всех болезней. Опираясь на этот популярный факт, можно с ответственностью заявить: подлинного модульного подхода к программированию ПО быть не может. Может быть только нечто похожее. Но отсутствие таблетки от всех болезней, не исключает наличия избавления от всех болезней - смерть. Я не думаю, что это может постигнуть нашу индустрию в целом, но некоторых частей определенно коснется.

За время эволюции, аппаратная среда росла в производительности. В некоторых аспектах она уже упирается в потолок, или близка к нему. Процесс производства аппаратных продуктов наладился, подешевел, стал более качественным. Не это ли называется эволюцией? Программная среда же как опиралась на принципы середины прошлого века, так и опирается. Да, появилось множество новых языков программирования, новых подходов, новых шаблонов проектирования. Однако смысл остался тем же. Не берусь оценивать данный факт. Вполне возможно, что это вовсе и не плохо. Может еще не пришло время?

Окунемся в девяностые года прошлого века. В 1994-1995 огромный успех платформы PHP дал все понять, что динамичному вебу быть. Microsoft, следуя за модой, запустила ASP. Принцип программных решений, написанных на этих платформах, был очень красивым. Вся логика реализовывалась на стороне сервера, клиентская же часть выступала терминалом, отображающим информацию. Прекрасное, красивое и элегантное решение. Но пользователь был недоволен. Ему хотелось интерактивности. Хотелось, чтобы всплывающее окно именно «всплывало». Тогда появилась поддержка скриптов на клиентской части, в браузерах. Если сравнивать с автономными приложениями на Delphi или C++ Builder пятнадцатилетней давности, то производительность этих «веб-монстров», слепленных из тысяч строк кода на JavaScript с примесью HTML и CSS, мала, чего не скажешь о сложности написания кода: пятнадцать лет назад модальное окно занимало одну-две строки, а сейчас сильно больше. Можно конечно подключить стороннюю библиотеку на несколько тысяч строк, но лишние пару сотен килобайт нагрузки на канал пользователя - роскошь. И вот так, одна за другой, тянутся головные боли современного разработчика, а также его ночные кошмары в виде Internet Explorer 6, на котором сидят богатые корпоративные клиенты.

В 1995 году появилась технология Java и начала активное наступление. Несмотря на маркетинговые усилия, Java не могла завоевать компьютеры пользователей. Причины тому были две: принудительность установки и дальнейшего обновления JRE и низкое быстродействие,

обусловленное кроссплатформенностью. Спустя 12 лет, компания Oracle заявляла о миллиарде устройств в мире, на которых установлена Java. Но мир уже двинулся в сторону «браузеризации». Та ниша, которую Java заняла не полностью, стала заниматься Flash-приложениями. Macromedia обошли грабли долгой и неудобной пользователю установки, но не смогла завоевать корпоративные сердца.

В этой части я не берусь судить правильность или неправильность мыслей автора, но в целом оспорить здесь ничего не могу.

В начале бума ООП одним из доводов «за», было утверждение «ООП позволяет увеличить количество кода, которое может сопровождать один среднестатистический программист». На данный момент мне не очень понятно, почему этот довод был доводом «за». Чем меньше кода пишет программист, тем меньше он допустит ошибок - мое сугубо личное мнение. Конечно, нельзя жертвовать читаемостью, но излишняя многословность - минус в нашей профессии. Но вернемся к ООП. Бьерн Страуструп ставил перед собой цель увеличить производительность труда программиста. Тем не менее, спустя некоторое время, как замечает автор, проблема все равно всплыла, порождая так называемый «Ад Паттернов». Сам термин мне категорически не нравится. Я не вижу ничего плохого, в следовании паттернам. Мне кажется это правильным, притом более чем. Любые попытки привести код в соответствие какому-то общепризнанному образцу - самая что ни на есть орфография. Автор приводит в пример факт из своей практики: «в рамках относительно автономного проекта мне пришлось интегрироваться с общим для нескольких групп Фреймворком ради вызова единственной функции авторизации пользователя: передаёшь ей имя и пароль, в ответ «да/нет». Этот вызов повлёк за собой необходимость явного включения в. NET-приложение пяти сборок. После компиляции эти пять сборок притащили за собой ещё более 30, большая часть из которых обладала совершенно не относящимися к безопасности названиями, вроде XsltTransform. В результате объём дистрибутива для развёртывания вырос ещё на сотню мегабайтов и почти на 40 файлов. Вот тебе и вызвал функцию…». Да-а-а, конечно же, поварами, заварившими это блюдо, состоящее из спагетти-кода и фрикаделек неумения (недостатка практики) и заправленное соусом сжатых сроков, являются именно ООП и «Ад паттернов», а никак не программисты, несомненно. Если же привести в пример какое-либо API современной мобильной операционной системы, то вы увидите: нет ничего более красивого и стройного, чем логика этого самого API. Я не берусь разбирать Android API по паттернам, но уверен, что с ними полный порядок.

Автор ругает C++, используя выражение «легко выстрелить себе в ногу». Тут и спорить нечего. C++ действительно требует хорошей подготовки и хорошей дисциплины, дабы не допустить непониманий внутри команды, или может даже внутри самого себя. Далее появились очень-очень объектно-ориентированные языки, такие как Java и C#, но проблемы с проектированием остались. Тем не менее ООП по сей день является самой используемой парадигмой современного программирования. Совпадение, привычка или общее заблуждение? Не знаю.

Начав с основы современного программирования, а именно ООП, автор двинулся к реляционным СУБД. Реляционная СУБД оперирует данными в виде реляционных структур. ООП же - в виде объектов. Это два коллинеарных направления. Но все равно они как-то уживаются в рамках одной программы, в рамках одного подхода. Как? Благодаря ORM. Однако, по словам автора, программисты не научились пользоваться ею, что повлекло за собой нагромождение непонятного кода и появление дополнительных, ненужных слоев абстракции. Но панацея нашлась. И даже три, «одна другой лучше»: разработке собственного проектора, изучение и понимание механизма работы ORM, или… переход в NoSQL. С ORM-системами я, к сожалению, знаком довольно мало, в связи с чем не могу оценить изложение автора.

Софтостроение - высокотехнологичный процесс. По причине развития общедоступных каналов связи работать с программами можно с удалённого терминала, в роли которого выступает множество устройств: от персонального компьютера до мобильного телефона. Значит ли это, что софт - услуга? Нет. Услуги физически неосязаемы и не поддаются хранению. классическая цепочка «производитель – конечный потребитель» подразумевает, что потребитель сам приобретает нужную программу и право на использование, сам её устанавливает и эксплуатирует.

Вы думаете, что большие ЭВМ вымерли или вымирают? Нет. Нам же казалось всегда, что вымирают и их век сошел на нет. Нам говорили «им пришли на смену новые технологии». А всегда ли «новые технологии» олицетворяют прогресс? Должно быть да, но нет. Еще 15-20 лет назад презентации на конференциях разработчиков были своеобразным мастер-классом, где на бета-стадии испытывалась реакция аудитории на предлагаемые изменения. Сегодня повестка дня состоит в постановке перед фактом новой версии платформы, показе новых «фишек» и оглашении списка технологий, которые больше не будут развиваться, а то и поддерживаться. [Далее автор углубляется в обзоры продуктов, что не представляется мне интересным. За исключением Windows Vista].

Windows 7 была в стадии release-candidat, а представители Microsoft открытым текстом стали предлагать отказаться от покупки своего флагманского на тот момент продукта – операционной системы Windows Vista и подождать выхода новой версии. По сути пользователям сообщили, что мы достаточно потренировались на вас и за ваши же деньги, а теперь давайте перейдём собственно к делу. Судьба Vista была решена – система фактически выброшена в мусорную корзину, так и не успев занять сколь-нибудь значительную долю парка «персоналок» и ноутбуков. Интересный факт: Билл Гейтс и Стив Балмер в конце 2000-х годов начали активно продавать свои доли в бизнесе Microsoft.

Основная задача проектировщика - поиск простоты. Красота в простоте. На этот счет высказывался Antoine de Saint-Exupery. С ним трудно не согласиться.

Краткий словарь для начинающего проектировщика, приведенный автором - абсурд. Звучит ровным счетом как высказывание 15-ти летнего подростка, не вышедшего из переходного периода, с присущим этому возрасту непоколебимым максимализмом.

Далее коснемся автоматизированной информационной системы (АИС). Любая АИС может быть рассмотрена с трех точек зрения проектировщика: логическое устройство, физическое устройство, концептуальное устройство. Это разделение видится мне обоснованным и логичным, но лить воду, как это делает автор, в данном случае я не намерен.

В ходе всех размышлений, опущенных мною, автор приходит к выводу, что наиболее прагматичным и удобным способом проектирования является «двухзвенка» - система с тонким клиентом. Спорить с данным заявлением бессмысленно и глупо - это мнение конкретного человека. Но для справедливости автор уточнил: «У каждой архитектуры есть свои недостатки и преимущества». Наиболее приятным и правильным, с моей точки зрения, изречением в данном пассаже было «Ищите возможности сократить путь информации от источника к пользователю и обратно». Разве не истина в полном смысле слова?

Живая история проектирования и воплощениям в жизнь проекта от автора книги и его коллег осталась за кадром. Я не посчитал необходимым конспектировать эту историю, несмотря на то, что она оказалась интересной. Согласен с мнением автора по поводу архитектуры: я тоже являюсь приверженцем так называемого «тонкого клиента». Не согласен со многим. С настолько многим, что перечислять не стану.

Существует два подхода к разработке корпоративных информационных систем (КИС): «от производства» и «от бухгалтерии». В первом случае ключевым моментом является планирование ресурсов. То есть решить задачу оптимизации. Во втором случае путь проходит через бухгалтерию, со всеми вытекающими отсюда последствиями. Именно таким образом и разрабатывалась КИС, о которой повествует автор.

Далее длинный рассказ автора об архитектуре проекта Ultima-S, конспектировать который не вижу смысла.

Считать трудозатраты на разработку архитектуры и ядра достаточно сложно. Так же трудно, если не сильнее, считать отдачу в дальней перспективе. Планирование - неблагодарный процесс. «Человек предполагает, Бог располагает». Я слышал очень большое количество мнений на сей счет, но все они были едины в одном: времени никогда не бывает много и чем больше надо, тем больше понадобится потом. Грустным концом продукта, о котором повествовал автор, было закрытие после более чем трех лет существования из-за убыточности.

Общие принципы решения типовых задач проектирования начали зарождаться еще давно. Но уже в середине 1990-х годов иногда возникали упоминания книги «Банды четырех» о приемах объектно-ориентированного проектирования, где была предпринята попытка их обобщения. Автор утверждает, что на практике толку от этой книги будет немного. Эта книга дала толчок для появления новых трудов, причем некоторые их них касались конкретного языка программирования, хотя сама цель шаблонов проектирования - обобщение.

Достаточно широко известна проблема принадлежности объектов друг другу с образованием иерархии. Касаемо сборщика мусора: очевиден плюс: программисту не стоит беспокоиться об освобождении памяти. В противопоставление этому (и другим) плюсу - производительность (и другие минусы).

Очень долгая преамбула с большим количеством примеров и экстраполяций, которая в итоге свелась к вполне понятному механизму бинарного семафора - мьютекса. Далее плавно перетекли к обсуждению путей оптимизации баз данных.

UML. Unified Modeling Language. Языком назвать сложно, а вернее неправильно. Несмотря на слово Language в названии. Да и Unified в названии - обман. UML вовсе не универсальный. Он скорее унифицированный, то есть «объединяющий». Но сути не меняет. Лично я не очень понимаю его важности и не вижу ему применения. Скорее всего это обусловлено моей неопытностью, но вместе с тем. Хотя UML-диаграммы сильно помогают в понимании паттернов проектирования. Что есть, то есть.

«Явление зависит от условий наблюдения. Сущность от этих условий не зависит.» Очень интересное заявление, которое, как мне кажется, невозможно подвергнуть критике. Именно этот факт различает гелиоцентрическую модель и геоцентрическую модель. Человечество пришло к гелиоцентрической модели, и теперь отрицание этого факта - абсурд. Но это же самое человечество когда-то пришло к геоцентрической модели, и отрицание было тем же самым абсурдом. Теперь же создан UML. Только в случае UML система не геоцентрическая, а заказчикоцентрическая. А что с альтернативами? Их не много. Наша отрасль сейчас в положении «лучше такой UML, чем никакой.»

Судить в этом отрывке я не стану, соответственно соглашаться или не соглашаться тоже. Мне показалось стройным сравнение геоцентрической модели с UML. Красиво и похоже на правду.

Первая часть повествования автора - пример случая, когда автоматическое управление памятью - не есть хорошо. Случай этот встретился в практике автора, касался компании, у которой было довольно интересное логирование данных. Также приводился пример другого проекта. Интересная жизненная история, недостойная конспектирования.

И, как ни странно, третья часть также была посвящена истории из жизни, и так же, как предыдущая, не удостоилась возможности быть законспектированной мной. Я прошу прощения, но действительно в этом отрывке из 23 страниц не смог найти ничего интересного.

В данном отрывке автор подчеркивает необходимость ревизии кода, на примере случаев из практики. Откровенно говоря, случаи удивительные и крайне неприятные: писать такой код надо постараться. Автор пришел к выводу, что ревизия кода очень полезна, но при условии, что ревизию будет проводить специалист, и ревизия проводится с самого начала написания кода. В целом я согласен с этим, хотя слышал много плохого от грамотных специалистов про ревизию кода.

«Быстро, качественно, дешево - выбери два критерия из трех». Это выражение очень четко описывает состояние в индустрии. Далее интересная статистика: среди проектов с объемом кода от 1 до 10 миллиона строк только 13% завершаются в срок, а около 60% свертываются без результата; в проектах от 100 тысяч до 1 миллиона строк кода 25% завершаются в срок и 45% свертываются без результата. Это удручающая статистика. Не знаю возможно ли исправить положение, но точно нужно искать выходы.

В любом софтостроительном процессе можно выделить 4 стадии: анализ, проектирование, разработка, стабилизация, что очень похоже на водопадную методологию. Водопад течет сверху вниз. Но на практике он может зациклиться из-за недостаточного планирования (анализа). Решается это обычной внимательностью к выбору подрядчика. Но ничего нельзя поделать с банальным непониманием подрядчика и заказчика. Хотя, по моему мнению, от этого не спасет никакая методология, автор пишет, что для снижения рисков было предложена спиральная модель софтостроения. Да, она слегка нивелирует расхождения в понимании между заказчиком и подрядчиком, но слабое место все же имеет: сложно определить продолжительность очередного витка. И опять панацея не найдена, но автор идет дальше, рассматривая методологию «снизу-вверх» с упорядочивающей моделью «сверху-вниз». Назвал он ее «гибкая». Спорить я не стану. Мне кажется, что здесь не в чем упрекнуть автора (а так хотелось).

Модульные тесты. Разработка модульных тестов - это тоже разработка. Считается, что полное покрытие кода тестами - залог успеха проекта. В отличие от тестов функциональных, модульные тесты требует переработки вместе с рефакторингом рабочего кода, что заставляет разработчиков меньше изменять код и больше делать надстройки. Плюс к этому: модульный тест, все же, не является залогом успеха. Прохождение кодом всех модульных тестов не гарантирует прохождение кодом функциональных тестов. Вывод: соизмеряйте затраты.

Идеи разрабатывать программы, минимизируя стадию кодирования я не очень понимаю. Мне кажется, что таким образом добиться качественного софта не представляется возможным. Палки себе же в колеса. Однако без таких идей, как мне кажется, прогресс шел бы несколько медленнее. В частности, в управляемой моделями разработке или в программной фабрике существует возможность сгенерировать код, который сразу будет работать, а дальнейшее проектирование можно продолжать взаимодействовать с готовой моделью, что действительно интересно. Мне очень понравилась аналогия с «мужиком с лопатой» и «экскаватором».

Далее автор привел пример работы с такого рода системой, что было очень интересно, потому что ничего подобного я не слышал и не видел. Это определенно меня заинтересовало, однако, на первый взгляд, мне все же не видится эта технология технологией, которой имеет смысл пользоваться всем. Но для генерации рутинного, одинакового кода - идеальное решение.

В жизни каждого более-менее сложного программного продукта наступает момент, когда продукт уже слишком сложный. Тогда он начинает жить своей жизнью. Это состояние нельзя назвать полным хаосом, но это не стройный порядок. В идеальном случае выходит так, что система в таком состоянии не долго мучает своих «поддерживателей» и умирает. И вот система, о которой автор

ведет речь, уже жила своей жизнью. Определить этот факт можно при помощи двух критериев: «Ты изменил чуть-чуть программу в одном месте, но вдруг появилась ошибка в другом, причём даже автор этого самого места не может сразу понять, в чем же дело», и второй: «Смотришь на чужой текст программы, и тебе не вполне понятен смысл одной его половины, а вторую половину тебе хочется тут же полностью переписать». Определив, что система именно из «этих», было принято решение бороться до последнего.

Очень сильно мне понравилось утверждение: если программист утверждает, что в этом месте программа «должна работать», значит он в этом месте программу не проверял. А если программист уже после обнаружения ошибки говорит «у меня работало», значит его нужно уволить. Очень ёмко, по моему мнению.

Современному софтостроению наука не нужна. Причиной тому я вижу капитализацию мира. Зачем вкладываться в науку, быть меценатом, если это принесёт столько же, а может и меньше, денег, чем если о науке даже не думать. Всем миром стал правил доллар. ("Доллар, доллар, доллар - грязная зелёная бумажка" В.В. Жириновский). В результате на первый план выходят люди, способные за короткое время и сравнительно небольшие деньги удовлетворить спрос на рынке. Такая тенденция сказалась и на технической литературе. Появление термина "гуглизация" яркое тому доказательство.

Мне показались интересными три правила Аникеева:

быть достаточно ленивым. Чтобы не делать лишнего, не ковыряться в мелочах;

поменьше читать. Те, кто много читает, отвыкают самостоятельно мыслить;

быть непоследовательным, чтобы, не упуская цели, интересоваться и замечать побочные эффекты.

Также очень интересным показалось заявление автора про лист бумаги и карандаш: "Прежде чем начать читать, возьмите листок бумаги, оптимально формата А4 – сложенный пополам, он удачно вкладывается в книгу, служа одновременно и закладкой. И запаситесь карандашом". Очень напомнило Александра Николаевича. Думаю, стоит попробовать.

Прочитав книгу до конца, я слегка изменил своё мнение об авторе. Он мне больше не кажется очень недовольным жизнью стариком лет 80, повидавшим очень многое и имевшим в виду чужое мнение. После почтения любой книги я чувствую легкую грусть. И эта книга, как ни странно, не стала исключением. Она оставила некий след в моих мыслях. И это совсем не так книга, о которой предостерегал автор. Но придя домой я её точно удалю.

Ярошевич Яна

Часть 1

- **Очень краткий экскурс**

Мир информационных технологий постоянно совершенствуется, но программист, по-прежнему, – это человек, способный заставить компьютер решать поставленное перед ним множество задач.

Но можно сказать, что существенно изменилось множество задач, которое необходимо решать программисту, методы решения этих задач и, естественно, технические возможности.

- **Специализация**

Хорошим специалистом, можно сказать, является тот, который может формализовать задачу, придумать алгоритм для ее решения, закодировать этот алгоритм и протестировать.

Вторую часть можно коротко описать известным выражением: «Как корабль назовешь, так он и поплывет».

- **Кто такой ведущий инженер, или как это было**

Описано соответствие названий подразделений, должностей, ролей в современных компаниях и организациях позднего СССР.

- **Метаморфозы**

Я, конечно, не знаю, какова ситуация в России, но в Беларуси, насколько мне известно, ситуация обстоит ровно также, как написано в источнике по поводу «прошлого», а именно в трудовых книжках в настоящее время до сих пор пишут «инженер-программист», добавляя «ведущий» или категорию для специалистов с высшим образованием, «техник-программист» для специалистов со средне-специальным образованием либо неоконченным высшим образованием.

Конечно, сейчас в Беларуси ведется работа по изменению названий должностей, например, была добавлена должность «специалист по тестированию ПО».

Во время обучения моих родителей в Беларуси был лишь один университет, который, в действительности, готовил специалистов, которые в будущем будут заниматься наукой, а в институтах готовили инженеров. Здесь я абсолютно согласна с автором книги.

- **О красоте**

Программирование нельзя целиком причислить ни к искусству, ни к ремеслу, ни к науке. Софтостроение на текущий момент – эклектичный сплав технологий, которые могут быть использованы как профессионалами технического творчества, так и профессионалами массового производства по шаблонам и прецедентам.

Будет ли человек творчески подходить к написанию софта, не зависит от уровня образования, от количества знаний, приобретенных в учебном заведении, - зависит лишь от того, является ли он «творческой натурой». Один сделает красиво, но недостаточно функционально, а второй наоборот.

- **6 миллионов на раздел пирога**

Человек, любящий свою работу, занимается ей отнюдь не только ради дохода. Даже если вообразить себе ситуацию, что такому человеку будут ежемесячно выплачивать некоторую сумму денег, достаточную для обеспечения его потребностей, то он все равно, скорее всего, будет стремиться к любимому делу.

- **Круговорот**

При необходимости автоматизировать часть работы на предприятии руководство заказывает разработку программного обеспечения, позволяющего выполнять данную часть работы, у сторонней IT-компании, к примеру. После получения готового ПО предприятие отказывается от персонала, который выполнял данную часть работы. После увольнения сотрудники меняют квалификацию и занимаются, например, разработкой ПО каждый раз в новой предметной области, при этом им приходится погружаться в незнакомую для них предметную область, тем самым фактически превращаясь в специалиста, выполняющего работу до автоматизации.

- **Масштабы и последствия**

Чтобы преуспеть в программировании, нужно соответствовать одному из трех пунктов:

1. Быть хорошим специалистом в своей области
2. Быть специалистом в некотором узком направлении
3. Быть специалистом еще в некоторой предметной области, кроме программирования

- **Профориентация**

Софтостроение на 90% находится в сфере услуг, поэтому взаимодействия типа «человек-человек» особенно важны в этой области. Отсюда следует логичный вывод: чтобы работать софтостроителем, нужно уметь работать с людьми, обладать навыком коммуникации, и без этого никуда.

- **Начинающим соискателям**

Автор попытался «расшифровать» некоторые фразы в объявлениях о вакансиях. На мой взгляд, его «расшифровки» отнюдь не всегда соответствуют действительности.

В общем случае, собеседования на должность софтостроителя происходят примерно одинаково. Человеку, пришедшему на собеседование, задают несколько «тестовых» вопросов на знание технологий и возможно дают небольшое практическое задание. Подчеркивается то, что по ответам на вопросы очень сложно определить уровень пришедшего. Делается акцент на то, что на тесты легко натаскаться, и ответить на все, не понимая значения почти всего сказанного.

- **Про CV**

Существует несколько основных правил, связанных с написанием резюме. Стоит быть кратким, излагать информацию максимально понятно, не преувеличивать и не врать, делать упор на технологии, стараться не делать ошибок. На мой взгляд, все эти советы являются весьма ценными, хоть и очевидными.

- **Про мотивацию**

Мотивация – исключительно внутренний механизм. Чтобы управлять им, необходимо залезать в этот самый механизм, в психику.

Стимуляция – это внешний механизм. Он основан на выявлении мотивов и последующем их поощрении или подавлении.

Управлять мотивацией, то есть целенаправленно изменять психологию и выстраивать набор стимулов – это «две большие разницы». Первое, по сути, требует изменения самих людей, второе – это использование имеющихся у них мотивов. Мотивировать же можно только свои поступки, но никак не образ действия окружающих.

- **Изгибы судьбы при поиске работы**

Поиск подходящей работы – дело нелегкое. Иногда стоит идти на многие уступки и пробовать. И, конечно, всегда стоит уважать персонал из кадровых служб!

Ответы на вопросы

- **Наиболее полезные пункты, по моему мнению:**

- О красоте.
  Этот пункт заставил задуматься о возможности при желании проявлять творчество, работая даже на заурядной должности.
- 6 миллионов на раздел пирога.
  В этом пункте описана идея безвозмездной любви к своему делу, что заставило задуматься, так ли оно на самом деле в моем конкретном случае
- Масштабы и последствия.
  В этом пункте описаны некоторые «формулы успеха», что является достаточно полезной информацией, на мой взгляд.

- **Не согласна со следующим:**
  - В пункте «Начинающим соискателям» были приведены некоторые «расшифровки» фраз из объявлений о вакансиях. На мой взгляд, автор «перегнул палку», и отнюдь не всегда его мнение соответствует действительности.
  - По моему мнению, в пункте «Метаморфозы» неверно приведен пример того, что указанное разделение было в прошлом. Уверена, что оно имеет место и сейчас, только немного с другим подтекстом.
  - Не до конца согласна с пунктом «Круговорот», создается впечатление, что он немного «притянут за уши».

Часть 2

- **Можно ли конструировать программы как аппаратуру?**

Можно сказать, что в софтостроении имеются относительно стандартные подсистемы: операционные среды, базы данных, вебсерверы, программируемые терминалы и тому подобное. Однако все они имеют достаточно сложную конструкцию.

Отсюда возникает вопрос, можно ли в софтостроении использовать схему конструирования программы из отдельных компонент.

Автор утверждает, что для аппаратуры используется модель конечного автомата. А в софтостроении такую модель можно использовать при двух основных условиях:

1. Программист ознакомлен с этой теорией
2. Количество состояний обозримо: они, как и переходы, легко определяются и формализуется И не стоит забывать об ограничениях технологий.

- **Безысходное программирование**

Безысходное программирование – программирование без исходников. Т.е. когда программисты пишут код, не имея доступа к исходному коду некоторой подпрограммы, компоненты или класса. Безысходным программированием можно также назвать использование библиотек, исходный код которых мы не знаем, с этим каждый программист сталкивается постоянно.

Однако, в таком случае, программисты не имеют никаких гарантий, что в любой возможной ситуации их софт поведет себя так, как надо. А цена ошибки велика, да и ответственность с разработчиков никто не снимал.

- **Эволюция аппаратуры и скорость разработки**

Автор утверждает, что с 1980-х годов производительность «железа» возросла на порядки, стоимость прилично снизилась, увеличилась надежность, однако принципы остались прежними. Также автор считает, что скорость разработки снизилась.

Есть статистика, показывающая, что количество работающих в софтостроении женщин росло до 1990-ых годов, после чего резко пошло на убыль. Автор утверждает, что это связано с начало массового использования принципов ООП.

Ответы на вопросы

- **Наиболее полезные пункты, по моему мнению:**
    - Можно ли конструировать программы как аппаратуру? Интересной показалась идея складывать программу из компонент, как дом из кирпичиков. Переходить от одного состояния к другому. С такой методой я согласна полностью.
    - Безысходное программирование
      Действительно, не всегда стоит доверять компонентам, взятых с каких-либо источников, особенно если у них нет доступных исходников. В таких случаях иногда стоит сделать эту часть работы самому.

- **Не согласна со следующим:**
    - В пункте «Эволюция аппаратуры и скорость разработки» автор утверждает, что по сравнению с 80-ми годами скорость разработки снизилась. На мой взгляд, его доводов отнюдь не достаточно. По сравнению с 80-ми годами сейчас в программировании ставятся абсолютно другие задачи, более трудоемкие и требующие большего внимания и времени, так как мы работаем сейчас с абсолютно другими технологиями. Поэтому считаю, что сравнивать скорость разработки некорректно в этой ситуации.
    - В пункте «Эволюция аппаратуры и скорость разработки» автор отмечает, что количество работающих в софтостроении женщин росло до 1990-ых годов, после чего резко пошло на убыль. Автор утверждает, что это связано с начало массового использования принципов ООП.
    - Категорично не согласна с автором в этом вопросе. Не считаю, что переход к ООП являлся сложным вообще для любого пола, просто взгляд на вещи в программировании был немного изменен. Не думаю, что женщины не справились с этой задачей, поэтому их количество начало падать. На мой взгляд, это могло произойти с ростом престижности работы программиста и количеством зарабатываемых средств. В связи с этим, мужчины просто могли начать «вытеснять» женщин с этого рынка, так как им свойственно стремиться к такой профессии больше в этой ситуации. Такое отношения автора к вопросу действительно заставило задуматься о качестве книги.

Часть 3

- **О карманных монстрах**

Автор приводит пример некоторой конторы, в которой имелась небольшая и простая система ведения заказов на рекламу для книжно-журнального издательства.

Утверждается, что 10-15 лет назад можно было для написания системы использовать Delphi/C++, PowerBuilder, Visual Basic, FoxPro, легкую клиент-серверную СУБД, и приложение было бы написано за 3-5 дней с написанием нескольких сотен строк прикладного кода.

В 2009 же году приложение было сделано с использованием .NET в трехзвенной архитектуре. Приложение занимает примерно 20 тыс. строк на C#, из них более половины являются техническими.

Возможно, лучше было бы прибегнуть к варианту 10-15-летней давности. Но механизм принятия решения базировался, во-первых, на том, что менеджеру необходимо было использовать только платформы и средства Windows, а, во-вторых, программист не умел опыта разработки вне шаблонов многозвенной архитектуры, поэтому не стал рисковать.

И такие решения принимаются в мире ежечасно.

- **ASP.NET и браузеры**

В свое время Microsoft не могла остаться в стороне, когда PHP стал развиваться, и выдала собственное решение под названием ASP (Active Server Pages), работающее только под Windows. Вскоре в браузеры тоже включили поддержку скриптовых языков.

В итоге исходная веб-страница стала включать в себя скрипты для выполнения вначале на сервере, а затем и на клиенте. Многие сотни строк каши из HTML, VBScript и клиентского JavaScript. Последующая эволюция технологии была посвящена борьбе с этой лапшой. Однако, несмотря на значительный прогресс, производительность разработки пользовательского интерфейса для веб-приложений в разы отстаёт от автономных приложений.

Достаточно быстро выяснилось, что разработка приложений, работающих хотя бы под двумя типами браузеров - задача не менее сложная, чем написание кода в автономном приложении на базе переносимой оконной подсистемы. А тестировать нужно не только под разными браузерами, но и под разными операционными системами. С учётом версий браузеров.

Поскольку отступать было поздно, решили ввести стандарты: К началу 2000-х годов установился фактический стандарт корпоративного веб-приложения: Internet Explorer 6 с последним пакетом обновления под Windows 2000 или Windows XP.

Под эти требования за 10 лет было написано множество приложений. А когда пришла пора обновлять браузеры, внезапно выяснилось, что их новые версии далеко не всегда совместимы с находящимися в эксплуатации системами.

Подводя итоги, автор утверждает, что людям, ответственным за выбор технологий, стоит всячески обосновывать необходимость использования веб-интерфейса в системе, принимая в рассмотрение другие пути.

- **Апплеты, Flash и Silverlight**

Когда начал распространяться язык Java, то больше всего его было в двух направлениях: полноценные desktop приложения и апплеты (приложения, имеющие ограничения среды исполнения).

В 2007 году Sun утверждала, что среда исполнения Java установлена на 700 миллионах персональных компьютеров, правда не уточнялась её версия. В декабре 2011 года уже новый владелец – корпорация Oracle – привёл данные о том, что Java установлена на 850 миллионах персональных компьютеров и миллиардах устройств в мире.

Тем не менее необходимость в кросс-платформенных богатых интерактивными возможностями интернет-приложениях никуда не исчезла. Эта ниша к началу 2000-х годов оказалась плотно занятой Flash-приложениями, специализирующимися на отображении мультимедийного содержания. Учтя ошибки Java, разработчики из Macromedia сделали инсталляцию среды исполнения максимально лёгкой в загрузке и простой в установке. Но такая специализация технологий оказалась непригодной для разработки корпоративных приложений.

К решению проблемы подключилась Microsoft. Первым «блином» в 2005 году стала технология ClickOnce развёртывания полноценных WinForms-приложений. Sun отреагировала молниеносно, добавив аналогичную возможность под названием Java Web Start.

Развитие WinForms было заморожено, и переход в. NET 3 к более общей технологии построения пользовательских интерфейсов WPF29, отличающейся универсальностью и большей трудоёмкостью в прикладной разработке, но позволяющей полностью разделить труд программистов и дизайнеров, что имело смысл в достаточно больших и специализированных проектах.

Побочным продуктом WPF стал Silverlight. По сути, это реинкарнация Java-апплетов, но в 2007 году, спустя более 10 лет, и в среде. NET. Кроме того Silverlight должен был по замыслу авторов составить конкуренцию Flash в области мультимедийных интернет-приложений. В отличие от WPF, Silverlight вызвал больший энтузиазм разработчиков корпоративных приложений. Однако, Silverlight принес множество разочарований позже.

- **Что не понравилось:**
    - По-моему, автор крайне однобоко смотрит на возможности веб-приложений, а именно на работу со встраиваемыми скриптами. На его взгляд, работа с такой технологией чрезмерно трудоемка, с чем я не согласна, так как сама имела с ней дело неоднократно, наравне со второй описанной технологией. Да, нужно предусматривать работу приложения во всех браузерах, но на настоящий момент это не составляет особого труда, так как IE был снят с поддержки, а это очень сильно облегчило задачу, так как программирование под другие браузеры отличается между собой лишь деталями.

Часть 4

- **ООП – неизменно стабильный результат**

В начале широкой популяризации ООП, происходившей в основном за счёт языка C++, одним из главных доводов был следующий: «ООП позволяет увеличить количество кода, которое может написать и сопровождать один среднестатистический программист». Приводились даже цифры, что-то около 15 тысяч строк в процедурно-модульном стиле и порядка 25 тысяч строк на C++.

Появилась такая проблема, как Ад 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 сложных и многоуровневых конструкций в стиле «новый адаптер вызывает старый» по мере так называемого эволюционного развития системы;
- Лоскутная интеграция существующих систем и создание поверх них новых слоёв API.

Код начинает изобиловать плохо читаемыми и небезопасными конструкциями. Последствия от создания Ада Паттернов ужасны не столько невозможностью быстро разобраться в чужом коде, сколько наличием многих неявных зависимостей.

Появились новые C-подобные языки: Java, затем C#. Когда многие технические проблемы были решены, оказалось, что ООП очень требовательно к проектированию, так и оставшемуся сложным и недостаточно формализуемым процессом.

Учебники по ООП полны примеров, как легко и красиво решается задачка отображения геометрических фигур на холсте с одним абстрактным предком и виртуальной функцией показа. Но стоит применить такой подход к объектам реального мира, как возникнет необходимость во множественном наследовании от сотни разношёрстных абстрактных заготовок.

Автор утверждает, что Объектно-Ориентированный Подход на практике в большинстве случаев превращает проект или продукт, переваливший за сотню-другую тысяч строк, в упомянутый Ад Паттернов, который, несмотря на формальную архитектурную правильность и её же функциональную бессмысленность, никто без помощи авторов развивать не может.

- **Сокрытие базы данных, или как скрестить ежа с ужом**

Реляционным СУБД удалось в 1980-х годах освободить программистов от знания ненужных деталей организации физического хранения данных, отгородиться от них структурами логического уровня и стандартизованным языком SQL для доступа к информации. Также оказалось, что большинство форматов данных, которыми оперируют программы, хорошо ложатся на модель двумерных таблиц и связей между ними.

На практике сложилась ситуация, когда программы пишутся в основном с использованием ООП, тогда как данные хранятся в реляционных БД. Не касаясь пока вопроса целесообразности такого скрещивания «ежа с ужом», примем ситуацию как данность, из которой следует необходимость отображения объектов на реляционные структуры и обратно.

Ввиду упомянутого отсутствия под ООП формальной теоретической базы эта задача, в общем случае нерешаемая, выполнима в случаях частных.

- **Как обычно используют ORM**

Автор утверждает, что достаточно часто разработчики предпочитают использовать ORM из-за недостатка знаний реляционных СУБД. Однако впоследствии оказывается, что собственный язык запросов ORM генерирует далеко не самый оптимальный SQL. При небольших задачах это можно простить, однако нужно по-хорошему задуматься, если речь идет о сложных запросах к СУБД.

- **ORM на софтостроительной площадке**

Автор рассказывает про то, как ему довелось работать консультантом на одной IT-фирме, разрабатывающей специальную систему документооборота для управления жизненным циклом товаров. Рассказывается о том, что не велось фактически никакой документации, частые релизы и связанный с ним аврал, работа в тесном и жарком помещении. О рефакторинге с каждым днем задумывались все меньше и меньше. Разработчики приходили к абсолютно неоптимальным решениям за счет незнания технологий и своей необразованности. При загрузке одного документа было насчитано порядка 20 тысяч (!) коротких SQL-запросов.

Когда автор предложил альтернативное более оптимальное решение для работы с СУБД – от него отказались, так как никто бы не смог потом разобраться в этом решении после ухода самого автора.

В итоге после долгих дней стресса с большими усилиями систему дотянули до сдачи, проблемы системы назвали особенностями, а команду частично распустили.

Часть 5

- **ВЦКП в облаках**

На заре массового внедрения АСУ на предприятиях в 1970-х годах советские управленцы и технические специалисты предвидели многое. А именно, сосредоточение ресурсов в вычислительных центрах, где конечные пользователи будут получать обслуживание по решению своих задач и доступ. Доступ по простым терминалам, в ту эпоху ещё алфавитно-цифровым. Такова была концепция ВЦКП – Вычислительного Центра Коллективного Пользования.

В 1972 году впервые прозвучали слова о государственной сети вычислительных центров, объединяющих региональные ВЦКП. Для чего, по большому счёту, и потребовалась унифицированная техническая база в виде ЭВМ ЕС и позднее СМ ЭВМ.

Проектировщики, техники и организаторы тогда занимались примерно тем же самым, чем занимаются сейчас продвигающие на рынок системы «облачных» вычислений, за исключением затрат на рекламу. Создавали промышленные вычислительные системы, отраслевые стандарты и их реализации.

В чем состояли просчеты советской программы ВКЦП? Проглядели стремительную миниатюризацию и, как следствие, появившееся обилие терминалов. Просчитались по мелочам,

не спрогнозировав развал страны с последующим переходом к состоянию технологической зависимости.

Программировать распределенное приложение сложнее, чем централизованное. Не только технически, но и организационно. Но в качестве выигрыша получаем автономию сотрудников, рабочего места и отсутствие необходимости поддержки службы в состоянии 24/7.

Наиболее очевидное применение децентрализации – автономные программы аналитической обработки данных. Мощность терминала позволяет хранить и обрабатывать локальную копию части общей базы данных, используя собственные ресурсы не заставляя центральный сервер накаляться от множества параллельных запросов к СУБД.

- **SaaS и манипуляции терминами**

На выходе софтостроения получаем продукт – программу, программный пакет, систему или комплекс.

По причине развития общедоступных каналов связи работать с программами можно с удаленного терминала, в роли которого выступает множество устройств: от персонального компьютера до мобильного телефона. Эксплуатацию же программ ведут поставщики услуг «в облаках» - современные ВЦКП. Это и есть “software as a service” – SaaS.

Но это не значат, что софтостроительные фирмы сейчас «производят услуги».

Во-первых, услуги не производят, а оказывают. И услуга от продукта отличается следующим:

- Услуги физически неосязаемы
- Услуги не поддаются хранению
- Оказание и потребление услуг, как правило, совпадают по времени и месту

Так что программа – продукт, как и многие другие.

- **От CORBA к SOA**

CORBA (Common Object Request Broker Architecture) – технологический стандарт разработки распределенных приложений, продвигаемый социумом OMG.

SOA – Service Oriented Architecture.

Предшественником современной СОА на базе вебслужб, с полным правом можно считать CORBA. Это сейчас задним числом обобщают подходы, представляя СОА как набор слабосвязанных сетевых служб, реализовать которые можно по-разному.

CORBA имела очевидные достоинства, прежде всего это *интероперабельность* и отраслевая стандартизация не только протоколов (шины), но и самих служб и общих механизмов – *facilities*.

Но CORBA 2, продержавшись несколько лет в основном потоке технологий, была отнесена на обочину. Выделим следующие причины:

- Стандарт поддерживался многими корпорациями, поэтому развитие требовало длительного согласования интересов всех участников. Некоторые из них продвигали свои альтернативы.
- Основным средством разработки оставался C.
- Запоздалая, объёмная и весьма сложная спецификация компонентной модели.

Постепенно появились веб-сервисы, которые из-за проблем с CORBA должны были стать новой платформой интеграции служб и приложений в гетерогенной среде. Концепции быстро придумали многообещающее название СОА.

Но и СОА не оправдала всех ожиданий. Современные заявления о том, что СОА не оправдала возложенных на нее надежд, свидетельствует о том, что и выбранная для нее модель веб-служб не стала решением проблем взаимодействия приложений в корпоративной среде. Ожидает ли нас новое пришествие CORBA в виде облегченной ее версии – покажет время.

- **Прогресс неотвратим**

Автор утверждает, что большие ЭВМ не вымирают, и приводит некоторую статистику.

Наиболее ходовым термином в софтостроении является «новые технологии». По умолчанию новые технологии олицетворяют прогресс. Но всегда ли это так?

- **NET**

NET меняются со скоростью, заметно превышающей сроки отдачи от освоения и внедрения технологий.

Кто же выигрывает в этой гонке кроме самой корпорации, пользующейся своим квазимонопольным положением:

- Услуги по сертификации (прямая выгода)
- Консультанты и преподаватели курсов (прямая выгода с некоторыми убытками)
- Программисты в целом (состояние неопределенности)
- Разработчики в заказных проектах (прямые убытки)
- Разработчики продуктов (косвенные убытки)

Часть 6

- **Краткий словарь для начинающего проектировщика**

Автор приводит забавный «словарь» начинающего проектировщика, то есть расшифровки слов коллег.

- «Это был плохой дизайн». Это спроектировано не мной.
- «By design» (так спроектировано). Ошибка проектирования, стоимость исправления которой уже сравнима с переделкой части системы.
- «Это не ошибка, а особенность (not a bug but a feature)». Прямое следствие из «by design».
- «Это может ухудшить производительность». Не знаю и знать не хочу ваши альтернативные решения.
- «Нормализация не догма». Потом разберёмся с этими базами данных, когда время будет.
- «Это наследуемый модуль». Этот кусок со многими неявными зависимостями проектировали достаточно давно, скорее всего стажёры.
- «Постановка задачи тоже сложна». Ума не приложу, откуда возникли эти десятки тысяч строк спагетти-кода.
- «Сроки очень сжатые». Мы давно забили на проектирование.
- «Наши модульные тесты покрывают почти 100 % кода». А функциональными тестами пусть занимается заказчик.
- «В нашей системе много компонентов». Установку и развёртывание системы могут сделать только сами разработчики.

- **Слоистость и уровни**

Разбираться в слоях и уровнях должны не только разработчики КИС, то есть «скелета», но и те программисты, которые будут наращивать на него свои приложения.

Определение автоматизированной информационной системы (АИС) складывается из трёх основных её компонентов: людей, информации и компьютеров.

Любая АИС – это люди, использующие информационную технологию средствами автоматизации. И КИС не исключение.

Любая автоматизированная информационная система может быть рассмотрена с трёх точек зрения проектировщика:

- Концептуальное устройство;
- Логическое устройство;
- Физическое устройство.

- **Концептуальное устройство**

Концептуальное устройство АИС состоит из трех слоев:

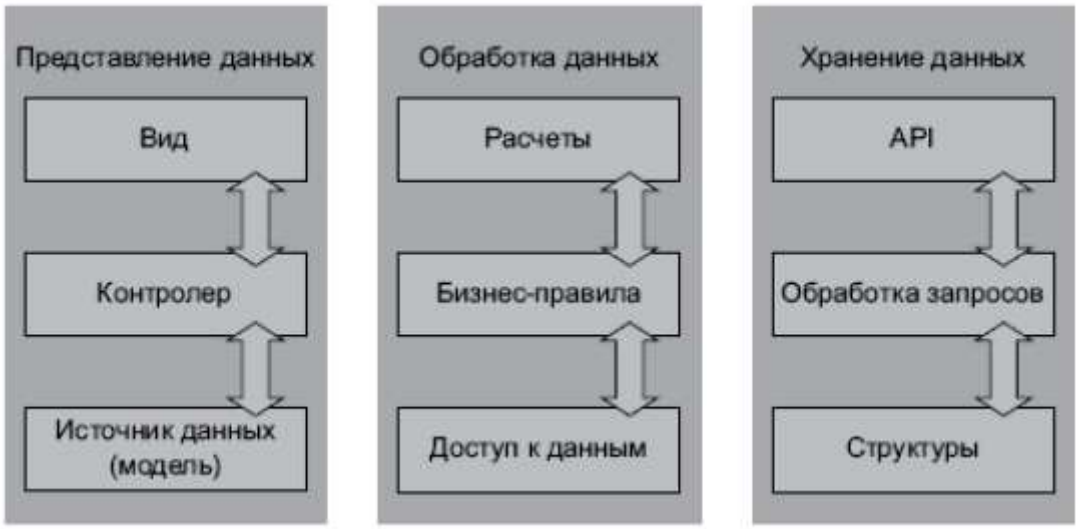
- Отображение данных
- Обработка данных
- Хранение данных

- **Логическое устройство**

Слои логического устройства не являются строго определенными.

Логическое устройство является предметом синтеза, на выходе стадии – технический проект.

Логическое устройство АИС в разрезе концептуальных слоёв может выглядеть, как на рисунке:



- **Физическое устройство**

Число звеньев системы определяется максимальным количеством процессов клиент-серверной архитектуры, составляющих цепочку между концептуальными слоями.

Тонким клиентом (thin client) традиционно называют приложение, реализующее исключительно логику отображения информации. В классическом варианте это алфавитно-цифровой терминал, в более современном – веб-браузер.

В противоположность тонкому, толстый клиент (rich client) реализует прикладную логику обработки данных независимо от сервера. Это автономное приложение, использующее сервер в качестве хранилища данных. В трёхзвенной архитектуре толстым клиентом по отношению к СУБД является сервер приложений.

Так называемый «умный» (smart client) клиент по сути остаётся промежуточным решением между тонким и толстым собратьями. Будучи потенциально готовым к работе в режиме отсоединения от сервера, он кэширует данные, берет на себя необходимую часть обработки и максимально использует возможности операционной среды для отображения информации.

- **Уровни**

Даже в простой программе типа записной книжки имеется минимум 2 уровня:

  - Уровень приложения, реализующий функционал предметной области.
  - Уровень служб, поддерживающих общую для всех разрабатываемых приложений функциональность. Например, метаданные, безопасность, конфигурация, доступ к данным и т. д.

В свою очередь, общие для многочисленных служб функции группируются в модули и соответствующие API уровня ядра системы.

С другой стороны, комплексная информационная система обладает множеством приложений, реализующих функционал нескольких предметных областей. В этом случае функциональность, связанная с их интеграцией и управлением, поднимается на уровень решения.

- **Совмещение**

Теперь если мы наложим слои системы на её уровни, то получим достаточно простую матричную структуру, позволяющую бегло оценить, какой из элементов необходимо реализовать своими силами или же адаптировать уже имеющийся готовый. Но, что более существенно, реализация разных подсистем может осуществляться независимо друг от друга после спецификации своих межуровневых и межслойных интерфейсов.

- **Многозвенная архитектура**

Итак, концептуальных слоёв в автоматизированной информационной системе всегда три: хранения данных, их обработки и отображения. А вот физических, их реализующих, может быть от одного, в виде настольного приложения с индексированным файловым хранилищем, до теоретической бесконечности.

Разумеется, у каждой архитектуры есть свои преимущества и недостатки. Если подходить к вопросу проектирования объективно, то выбор в каждом конкретном случае вполне рационален.

Сегодня доля специалистов, имеющих опыт в системном проектировании, в общем потоке становится всё меньше. Поэтому декомпозиция и выбор архитектуры часто проводится по принципу «прочитали статью, у этих парней получилось, сделаем и мы так же».

В итоге между экраном конечных пользователей и запрашиваемыми ими данными образуется толстая прослойка, которая на 80 % занята совершенно пустой работой по перекачиванию исходной информации из одного формата представления в другой. Растёт время отклика системы. Пользователь нервничает и справедливо обвиняет в этом программистов.

Кто не хочет – ищет причины, кто хочет – средства. Ищите возможности сократить путь информации от источника к пользователю и обратно. В конце концов, архитектура служит для эффективной реализации системы, а не освоения бюджета и вовлечения в команду очередных выпускников курсов переквалификации.

- **История нескольких #ifdef и включения в этот пункт**

Слишком сложно сделать краткий конспект этих пунктов, так как, на мой взгляд, почти все мелочи повествования важны.

- **Наиболее полезные пункты, по моему мнению:**
  - История нескольких #ifdef
  - Краткий словарь для начинающего проектировщика (заставил улыбнуться)
  - Уровни

Часть 7

- **Ultima-S – КИС из коробки**

В 1996 году компания «Ниеншанц» приступила к разработке не просто третьей версии внутренней системы управления предприятием, а к созданию тиражируемого коробочного продукта. Назвали новый продукт «Ultima-S».

Существует два основных подхода к разработке КИС, условно называемых «от производства» и «от бухгалтерии». В первом случае функциональным ядром системы становится планирование ресурсов производства. Альтернативный путь проходит через бухгалтерию. Большинство известных автору разработок КИС в России 1990-х годов шло «от бухгалтерии».

Поскольку основными потенциальными клиентами Ultima-S были именно оптово-розничные торговые компании, то функциональная архитектура системы базировалась на подходе «от бухгалтерии».

С учётом предполагаемого тиражирования, развивать продукт было решено в следующих направлениях:

- Минимизация числа звеньев;
- «Утончение» клиентского приложения, в идеале до уровня веб-браузера;
- Использование промышленной СУБД для реализации бизнес-логики;
- Реализация некоторых механизмов ООП для упрощения разработки прикладными и сторонними разработчиками методом надстраивания новых классов.

Синтезом вышеназванных приоритетов явилась двухзвенная архитектура с тонким клиентом.

В качестве базовой абстракции был выбран документ. То есть все объекты в системе – это документы, относящиеся к какому-либо их классу. Каждый документ хранился в одной из папок, составляющих иерархию, напоминающую вид обычного проводника Windows.

В рамках механизма ООП, надстроенного над процедурным, поддерживалось одиночное наследование реализации. Вместо формальных конструкторов, деструкторов и методов основу составил механизм событий, вызывающий процедуры-обработчики в порядке, зависящем от иерархии классов. Вся разработка сосредоточилась в одном звене и среде на декларациях классов, свойств-операций и на реализующих обработку событий хранимых процедурах.

Тонкий клиент отображал в динамике результаты обработки события сервером. Большинство экранных форм, таким образом, формировалось автоматически, программист только объявлял в процедуре соответствующие поля ввода и сетки. Для специфичных случаев расположения элементов управления было необходимо визуально проектировать форму либо во встроенном в приложение редакторе, либо в среде Visual Basic, загрузив затем описание формы на сервер.

Техническая архитектура – важный, но не единственный столп разработки тиражируемых продуктов. Никакая широта технической мысли не поможет, если на платформе реализуются неадекватные по сложности задачам решения, неполные или противоречивые требования. К сожалению, эта проблема не была решена и во второй версии прикладного обеспечения системной платформы.

По мнению коллеги автора, проектированием системы на самом деле должен заниматься не технарь, а маркетолог, так как система – это товар, удовлетворяющий потребности, а маркетолог к этому ближе. Он считает, что если бы разработчики Ultima-S, до того как взяться за написание кода, провели тщательное изучение архитектур аналогов на рынке, все было бы иначе.

Считать трудозатраты на разработку архитектуры и ядра достаточно сложно. Ещё труднее считать отдачу в дальней или даже средней перспективе. Если при разработке заказного проекта в первой версии приоритетом может быть именно ближняя перспектива, то в продуктовой разработке такой подход неизменно приводит к необходимости полной переделки второй версии.

Критическая масса клиентуры в проекте, выводящая его на прибыльность, не была достигнута, что послужило формальной причиной закрытия после более чем трёх лет существования. Но не прекращалась поддержка некоторых клиентов.

- **Нешаблонное мышление**

Зачастую, ещё вчерашние новички, научившись достаточно элементарным вещам, любят порассуждать о том, что изобретение велосипедов – пустое дело. По представлениям автора на воспроизведение большинства из приводимых в книжке GoF «велосипедов» в конкретных случаях у любого специалиста с навыками абстрактного мышления вряд ли должно уходить более часа. Это заметно быстрее изучения самой книги, ее осмысления и, наконец, осознания тех моментов, когда, собственно, надо применить абстрактный слепок в реальной жизни

Факт защиты одним из авторов научной диссертации по теме книги заставил в очередной раз автора призадуматься о степени проникновения современной теоретической мысли в практику софтостроения. По представлениям автора, книга имела к науке весьма опосредованное отношение и являлась скорее чем-то вроде поваренной книги.

Однако книга все-таки дала всплеск, и по воде пошли круги. Начали появляться подобные книги со своими велосипедами в определенных языках.

Однажды автор попытался объяснить программисту, увлекшимся шаблонами, что простой пример может иметь целый букет решений вместо одного шаблонного, из которых можно выбрать оптимальный, однако не встретил понимания.

- **Обобщение**

Откажитесь от термина «наследование», который искажает смысл действий. Используйте «обобщение».

Основное правило заключается в том, что обобщаемые классы должны иметь сходную основную функциональность.

Почему шаблоны «вдруг» стали ненужными? Потому что требуется только обобщение. Корректно проведя обобщение, вы автоматически получите код и структуру, близкую к той, что вам предлагают зазубрить и воспроизводить авторы разнообразных учебников шаблонов.

Основная ошибка – обобщение по неосновному функциональному признаку. Например, когда обобщают сотрудников и пользователей.

Ошибка приводит к построению иерархии по неосновному признаку. Когда внезапно найдётся ещё один такой признак, возможно, более существенный с точки зрения прикладной задачи, обобщить больше не удастся: придётся ломать иерархию или делать заплату в виде множественного наследования, недоступного во многих объектно-ориентированных языках. Или пользоваться агрегацией.

Не увлекайтесь обобщением. Ошибки тоже обобщаются и уже в прямом смысле этого слова наследуются. Исправление по новому требованию может привести к необходимости сноса старой иерархии, содержащей ошибки.

Глубина более двух уровней при моделировании объектов предметной области, вероятнее всего, свидетельствует об ошибках проектирования.

- **Про сборку мусора и агрегацию**

Сборщик мусора, он же GC – garbage collector в средах программирования с автоматическим управлением памятью. Наиболее очевидное преимущество – программисту не надо заботиться об освобождении памяти. Хотя при этом все равно нужно думать об освобождении других ресурсов, но сборщик опускает планку требуемой квалификации и тем самым повышает массовость использования среды. Но за все приходится платить.

Нужно взять за правило, что контейнер всегда управляет своими объектами. Поэтому обращаться к его внутренним объектам нужно только через интерфейс самого контейнера.

Сборщик мусора на некоторое время может скрыть логические ошибки проектирования интерфейсов, что является его существенным недостатком.

- **Наиболее полезные пункты, по моему мнению:**
  - Думать головой
  - Нешаблонное мышление

Часть 8

- **Журнал хозяйственных операций.**
  - Антишаблон «Таблица остатков»

Начинающий проектировщик рассуждает так: мне нужно текущее количество товара в наличии, а все движения, в результате которых эта цифра и появилась, можно оставить в стороне, а то и прямо в первичных документах.

Представьте, что в документ недельной давности вкралась ошибка. Её исправили, причём пересчитанные текущие остатки по-прежнему неотрицательны. Значит ли, что они неотрицательны и на каждый день прошедшей недели? Разумеется, нет.

Проектировщик приходит к неутешительному выводу: даже для фактических операций нужно считать остаток по истории (журналу). Не говоря уже о резервировании товара, где ситуация меняется гораздо быстрее: то тут отменили, то там подтвердили. Но считать по журналу:

- Может быть долго;
- Необходимо защитить считанные значения, чтобы при последующей записи не возникло «минусов».

Последнее, по сути, это и есть та самая сериализованная транзакция. Она гарантирует, что считанные значения не будут изменены другой транзакцией. То есть продажа не будет давать отрицательный остаток, если между операцией расчёта остатка и расхода вклинится другой.

Но почему нельзя просто блокировать всю таблицу-журнал, а надо использовать какие-то хитроумные транзакции с непонятным уровнем изоляции? Ответ прост – иногда транзакции могут не пересекаться, тогда в этом нет необходимости.

Совет от автора: учите сиквел, транзакции, уровни изоляции, и будет ваша система быстрой и надёжной. И не только в примере с учётной системой, но и вообще по жизни. Опытным же разработчикам есть поле для оптимизации и дальнейшего

совершенствования решений, включая альтернативные подходы.

- **UML и птолемеевские системы**

В UML должно настораживать уже самое первое слово – «унифицированный». Не универсальный, то есть пригодный в большинстве случаев, а именно унифицированный, объединённый.

Несколькими экспертами был создан сборник, содержащий наиболее удачные нотации из их собственной практики. Сборник стали продвигать в массы, а за неимением лучшего, и в стандарты. Стандарт получился несколько странным. В его названии присутствует слово «моделирование». Модель от картинки в графическом редакторе отличается наличием базового формализма, позволяющего проверить созданную конструкцию на непротиворечивость, а если повезёт, то и на полноту. Очень простой, но верный признак проблемы – отсутствие в инструменте моделирования команды «Проверить».

Графические образы – это, конечно, аналог слов, но не всякий набор слов является языком. Поэтому UML до языка ещё далеко. Язык, даже естественный, можно автоматически проверить на формальную правильность. Входящий в UML OCL хоть и является языком, но выполняет лишь малую часть работы по проверке, и не собственно модели, а её элементов. А о способностях проверить модель написано выше.

Поэтому использование UML имеет две основные альтернативы, напрямую зависящие от целей:

- Цель – использовать «как есть». Не заниматься вопросами целостности, ограничившись рисованием частей системы в разных ракурсах. Если повезёт, то часть кода можно будет генерировать из схем.
- Цель – использовать для моделирования и генерации кода. Придётся создать свои формализмы, соответствующие моделируемой области. В самом минимальном варианте – использовать в принудительном порядке стереотипы и написанные руками скрипты и ограничения для проверки непротиворечивости такого использования

С картинками в UML сложилась некоторая неразбериха вкупе с излишествами. Например, диаграммы деятельности (activity diagram), состояний (state machine diagram) и последовательностей (sequence diagram), а также их многочисленные подвиды по большому счёту описывают одно и то же – поток управления в программе.

В UML рекомендуется использовать комментарии для отражения требований к системе, а сами прецеденты неформальны. При отсутствии проработанной системы стереотипов приходится, например, отделять приходные документы от расходных с помощью раскраски.

Прецеденты, как и геоцентрическая модель, могут удовлетворительно описывать явление, не касаясь его сущности. Всякий раз разработчики очередного корпоративного приложения создают свою «птолемеевскую систему», которая при попытке примерить ее на другую фигуру внезапно оказывается неподходящей и нуждается в серьёзной перекройке.

Подобно уточняющим геоцентрическую модель эпициклам, на диаграмму накручиваются всё новые прецеденты. Видимо, поэтому в RUP рекомендуют не увлекаться наворачиванием прецедентов друг на друга (avoiding functional design). Исходя из этого совета можно легко

перейти в иную крайность, увидев «альтернативу» в том, чтобы побольше писать кода для заказчика и поменьше строить моделей.

Альтернатив, к сожалению, немного. Об основной – создании собственных предметно-ориентированных языков, метаязыков – уже сказано, и не раз, в том числе в рамках описаний конкретных систем. Далеко не факт, что для этого будет нужен UML хоть в каком-то виде. Отрасль же в очередной раз оказалась в интересном положении, когда «лучше такой UML, чем никакой». Слишком часто формулировка «лучше плохой, чем никакой» стала широко применяться, что настораживает. Но прогресс, развивающийся по законам бизнеса, неотвратим, даже если слово «прогресс», при наличии на то оснований, заключать в кавычки.

Из положительных моментов UML и поддерживающих его сред необходимо отметить, что в ситуации, когда задача складывается, как мозаика, из обрывочных сведений представителей заказчика, прецеденты помогут хоть как-то формализовать и зафиксировать неиссякаемый поток противоречивых требований. Неважно, что мозаика может, скорее всего, не сложиться в стройную картинку, но при отсутствии экспертизы в предметной области риски недопонимания и недооценки снижаются. Наиболее проработанная часть UML – диаграмма классов – при определённых усилиях и дисциплине программистов, конечно, поможет вам автоматизировать генерацию части рутинного кода приложения.

Большой интерес представляют так называемые «двусторонние» инструменты (two-way tools), тесно интегрированные со средами программирования и позволяющие непосредственно во время разработки оперировать диаграммами с одновременным отражением изменений в коде и, наоборот, импортировать меняющийся руками код в модель.

Часть 9

В одной крупной электрической компании имелась база данных, собирающая эксплуатационную и прочую информацию за несколько лет от разнообразных датчиков и устройств в распределительной сети национального масштаба. Размер хранилища составлял около трех терабайт информации за несколько фиксированных лет эксплуатации.

Пикантность ситуации состояла в том, что база данных была организована в виде двоичных файлов собственного формата. К великому счастью, программисты системы знали область, в которой работают, и добросовестно поддерживали проектную документацию в актуальном состоянии.

В задачу автора входило предпроектное обследование, начались опыты с новыми технологиями. Начали с прототипа конвертера, извлекающего данные из старых форматов и распределяющего их по таблицам промышленной СУБД, в качестве которой выступал SQL Server. Ввиду планируемой общей разработки исключительно на C# решили писать прототип именно на этом языке.

Быстро выяснилось, что побитовые операции с типами данных разной длины не являются сильным местом языка и платформы в целом. Команду автора настигла расплата за автоматическое управление памяти.

Второй проблемой стала база данных. На логическом уровне структура никаких сложностей не представляла, но трудность представлял уровень физический. Втиснуть число-признак, кодируемое тремя битами, в те же три бита на уровне СУБД стандартные средства не позволят. Минимальный размер – байт. Использовать нестандартные битовые поля в принципе можно, но тогда не было бы поддержки ссылочной и прочей целостности, возможности использовать SQL для выборок без переупаковки значений в битовые структуры.

При использовании минимальных по размеру стандартных типов данных прогнозируемый размер базы данных без учёта дополнительных индексов превысил 6 терабайт, то есть минимум

удвоился по сравнению с двоичными файлами. Сжатие данных несколько скрасило ситуацию, полученные 4,5 терабайта выглядели уже неплохо.

Третьим пунктом была загрузка данных из C#-приложения в базу данных. Поскольку интенсивность вставки из прибывающих от датчиков текстовых файлов составляла почти 400 тысяч записей каждые 10 минут, то, разумеется, приложение должно было справляться с этой задачей за меньшее время. Оптимальной, как и следовало ожидать, оказалась пакетная загрузка, игнорирующая ограничения целостности РСУБД.

Однако, автор совсем не уверен, что еще через 10 лет новые подрядчики, вынужденные в очередной раз переписывать систему, найдут документацию в том же полном виде, в котором нашла её команда автора.

Часть 10

- **Code revision, или Коза кричала**

Ревизия кода, несомненно, весьма полезная процедура, но как минимум при двух условиях:

- Эта процедура регулярная и запускается с момента написания самых первых тысяч строк;
- Процедуру проводят специалисты, имеющие представление о системе в целом. Потому что отловить бесполезную цепочку условных переходов может и компилятор, а вот как отсутствие контекста транзакции в обработке повлияет на результат, определит только опытный программист.

Качество кода во многом зависит от степени повторного использования, поэтому автор приводит простой и доступный способ проверки того, не занимается ли команда программистов копированием готовых кусков вместо их факторизации. Для этого регулярно стоит делать сжатый архив исходников, например zip с обычным коэффициентом компрессии, и оценивать динамику роста его размера относительно количества строк. Если размер архива растёт медленнее, чем количество строк, это означает рост размера кода за счёт его копирования.

- **Наживулька или гибкость?**

Американский специалист по методологиям софтостроения Кэпер Джонс в своей книге приводит весьма удручающие статистические данные, например:

- 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.

Если стадии, органично совпадающие с концептуальным, логическим и физическим дизайном системы, расположить иерархически без обратных связей, то получим классическую схему «водопад», исторически считающуюся первой методологией в софтостроении.

С увеличением сложности реализуемой системы анализ и следующее за ним проектирование начинают занимать всё больше времени. Постепенно обнаруживаются новые детали и

подробности, изменяются требования к системе, возникают новые сопутствующие задачи, расширяющие периметр. Приходится, не отдавая проектную документацию в разработку, возвращаться к анализу. Возникает риск зацикливания процесса без конечного выхода какого-либо программного обеспечения вообще.

Но если проблема зацикливания на требованиях может быть успешно решена выбором подрядчиков, уже имевших опыт в построении систем данного типа, то другая, гораздо более значимая проблема несовпадения взглядов заказчика и подрядчика на казалось бы одни и те же вещи стабильно проявляется с ростом проекта.

Для снижения рисков такого рода в конце 1980-х годов была предложена спиральная модель софтостроения. Ключевой особенностью спиральной технологии является прототипирование. В конце каждого витка после этапа стабилизации заказчик получает в своё распоряжение ограниченно работающий прототип целой системы, а не отдельных функций. Основная цель прототипа состоит в максимально возможном сближении взглядов заказчика и подрядчика на систему в целом и выявлении противоречивых требований.

Слабым звеном в спиральной методологии является определение длительности очередного витка, его стадий и соответствующее выработанному плану управление ресурсами. Тотальный анализ и проектирование вырождают спираль в водопад, тогда как формальный подход, «для галочки», выдающий бесполезные для программистов спецификации, превращает спираль в бесконечный цикл, прерываемый управленческим решением. Поиск компромисса в такой ситуации – трудная задача многокритериального выбора, в большой степени зависящая от опыта и здравого смысла руководителей рабочих групп и проекта в целом.

Ключевой особенностью гибкой методики является наличие владельца продукта, который лучше всех знает, что должно получиться в итоге. Именно владелец, за рамками собственно гибкого процесса, проводит анализ и функциональное проектирование, подавая команде на вход уже готовые пачки требований. Размер пачки должен укладываться в интеллектуальные и технологические возможности разработчиков, которым предстоит осуществить её реализацию за одну итерацию.

В итоге мы получаем знакомую софтостроительную схему «снизу-вверх», появившуюся на свет гораздо раньше «водопада».

Обе методологии («водопад» и «гибкая») зацикливаются с увеличением сложности проекта. Только «водопад» замыкается на тотальном анализе и проектировании, а гибкая методика «уходит в себя» на разработке и стабилизации. Из этого следует вывод, что успешно выполненный в «водопадной» схеме проект может быть также в большинстве случаев выполнен в «гибкой» разработке. И наоборот, если не касаться вопроса людских ресурсов. Дело в масштабе.

Современная тенденция – огромное число некритичных проектов, программ и систем-пристроек к основной КИС. Масштаб проектов небольшой (сотни тысяч строк кода), заказчик точно не знает, что хочет получить в итоге, а подрядчик не имеет опыта в данной предметной области, если вообще имеет хоть в какой-то. Для такой ситуации привлечение команд с имеющими требуемый опыт квалифицированными специалистами и технологиями предметно-ориентированных языков или разработки по моделям маловероятна, поэтому пусть уж лучше итеративная методология «наживульки», чем никакая, как оно зачастую бывало в эпоху штатных отделов АСУ.

Ответы на вопросы

- **Наиболее полезные пункты, по моему мнению:**
    - Все пункты были занимательными. Особенно стоит отметить тот факт, что на темы, поднимаемые в пунктах, мы уже успели написать доклады. Так что все было

понятно, и интересно было узнать точку зрения автора на темы, что мы поднимали сами.

Часть 11

- **Тесты и практика продуктового софтостроения**

Если изначально неизвестно, что выстроится в итоге, дом или коровник, то подпорки у его стен должны быть в любом случае.

Разработка модульных тестов – это тоже разработка. Для 100 % покрытия потребуется примерно столько же времени, сколько и на основную работу. А может, и больше, смотря как подойти к делу. По наблюдениям автора, соотношение объёмов рабочего и тестирующего кода примерно 1 к 2.

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.

Наконец, модульный тест – это самый нижний уровень проверок. Прохождение серии модульных тестов вовсе не гарантирует, что пройдут функциональные тесты.

Вывод: следует соизмерять затраты на создание и поддержу автоматизированных модульных тестов с бюджетом проекта и располагаемым временем.

- **Говорящие изменения в MSF и выключатель**

Microsoft Solution Framework – методология софтостроения, опирающаяся на практики Microsoft.

В своей текущей редакции MSF 4.0 была разделена на два направления: MSF for Agile Software Development и MSF for CMMI Process Improvement.

Соответственно, одно из нынешних направлений MSF предназначено для достижения зрелости процессов софтостроения или дальнейшего улучшения процессов уже более-менее зрелых организаций. А второе для гибкой разработки.

Ключевой особенностью системы в Тойоте является принцип «дзидока», означающий самостоятельность людей в управлении автоматизированной производственной линией. Если рабочий видит нарушение качества продукции или хода процесса, он имеет право, повернув соответствующий рубильник, остановить всю линию до установления причин дефектов и их устранения.

Качество программного продукта – многозначное и сложное понятие. Производственная культура – ещё более сложное. В одном можно быть уверенным: ни о какой культуре софтостроения не может идти и речи, если любой программист из коллектива не способен остановить бессмысленный циклический процесс для выяснения, почему по историям заказчика потребовалось обобщать четырёхногих коров и обеденные столы.

- **Приключения с TFC**

TFC = Team Foundation Server.

Заклинания, «гугление», помощь консультантов по TFS и даже переписка с Microsoft применялись многократно во время работы автора с TFC. Предложение автора использовать SVN не захотели понять. В то время, как TFC требовал много дополнительных компонент, SVN не требовал ничего.

Далее TFC принес еще много головной боли автору и его команде.

- **Программная фабрика: дайте мне модель, и я сдвину Землю**

В управляемой моделями разработке (УМР) и в программной фабрике в частности наиболее интересной возможностью является генерация кода, скомпилировав который, можно сразу получить работающее приложение или его компоненты. Мы проектируем и сразу получаем нечто работающее, пусть даже на уровне прототипа. Уточняя модели, мы на каждом шаге имеем возможность видеть изменения в системе. Проектирование становится живым процессом без отрыва от разработки.

- **Лампа, полная джинов**

Метафора системы достаточно проста: хочешь генерировать код компонента или слоя – попроси об этом соответствующего «джинна» в форме стандартного «заклинания». Джинны, как им и положено, живут в лампе.

Переходя к техническим терминам, программист описывает задачу в терминах логической модели, представляющей собой набор сущностей, их атрибутов, операций и связей между ними. Язык создан на основе XML, поэтому делать описания можно непосредственно руками в обычном текстовом редакторе.

Модель в виде XML-файлов поступает на вход «заклинателю» – входящей в состав пакета консольной утилите. 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, касающуюся не только самих джиннов, но и, например, таких настроек, как правила именования в конкретном слое системы.

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. Конечно, даже теоретически такое восстановление не может быть полным из-за разницы в семантике, но большую часть рутинной работы оно выполняет. Проведя один раз импорт, далее мы редактируем, структурируем модели и продолжаем работать только в обычном цикле изменений «через модель».

За джиннами следуют конфигурации слоёв. Если для домена и служб можно пока ограничиться спецификацией базового пространства имён, то для слоя хранения, особенно при поддержке более чем одной СУБД, необходимо указать дополнительные ограничения вроде максимальной длины имён. Заключительная часть конфигурации представляет собой описания шаблонов.

Теперь, если запустить «заклинатель» с параметром файла конфигурации проекта и не будет обнаружено ошибок, на выходе мы получим инициализированные структуры баз данных и готовые к компиляции файлы.

- **Слой хранения (СУБД), Слой домена (NHibernate), Слой веб-служб и интерфейсов доступа (ServiceStack)**

Описание слоев.

- **Остановиться и оглянуться**

Если рассмотреть метрики относительно небольшого проекта, то 40 прикладных сущностей в модели, состоящей примерно из 600 строк XML-описаний, порождают:

- Около 3 тысяч строк SQL-скриптов для каждой из целевых СУБД;
- Порядка 10 тысяч строк домена;
- 1200 строк XML для проекций классов на реляционные структуры (таблицы);
- Около 17 тысяч строк веб-служб и интерфейсов.

Таким образом, соотношение числа строк мета-кода описания модели к коду его реализации на конкретных архитектурах и платформах составляет около 600 к 30 тысячам или 1 к 50.

Это означает, что оснащённый средствами автоматизации программист с навыками моделирования на этапе разработки рутинного и специфичного для платформ/архитектур кода производителен примерно так же, как и его 50 коллег, не владеющих технологией генерации кода по моделям.

Ответы на вопросы

- **Наиболее полезные пункты, по моему мнению:**
  - Тесты и практика продуктового софтостроения
  - Остановиться и оглянуться

Часть 12

- **Хаос наступает внезапно**

«Баг» - это жучок-вредитель в программе, то есть ошибка, аномалия, сбой.

Автор рассказывает про очередной свой проект, на этот раз французский, на нем он помогал править «баги».

Сергей отмечает, что качество кода было не самое лучшее: обилие возвратов из разных мест одной функции и очень похожими кусками кода в одноимённых перегруженных функциях, написанных методом копирования текста через буфер обмена, «copypaste».

Система, с которой он начал работать, успешно прошла порог увеличения сложности еще несколько месяцев назад и жила отдельной от авторов жизнью. Контора решила пойти по следующему пути: выдать желаемое за действительное, побольше маркетинга, шуму, «подцепить» несколько заказчиков и на их деньги попытаться выпустить новую версию. И вот, одна фирма решилась на опытное внедрение, ее примеру последовала еще одна.

- **Что-то с памятью моей стало**

Автор описывает структуру конторы, в которой работал.

Двое основателей, каждый отвечает за различные сферы.
Появились двое русских программистов, на которых легла вся системная разработка и поддержка последнего уровня. Из остальных была составлена группа прикладных программистов.

Итого человек 15.

Один из основателей очень близко к сердцу каждый раз воспринимал все неполадки системы. Сначала вдохновлялся общением с заказчиками, но затем каждый раз программисты опускали его с небес на землю.

В какой-то момент разработчики поняли, что ядро с немереным аппетитом кушает оперативную память сервера. Утечки памяти составили около одного гигабайта в сутки.

Тогда разработчикам, по причине недостатка времени, пришлось работать брать овертаймы, работать ночью. Им удалось ликвидировать утечки памяти, но были еще некоторые проблемы.

- **Три дня в IBM**

У одного из заказчиков был сервер от фирмы IBM, под управлением ОС AIX-IBM-овского варианта UNIX.

Второй русский разработчик, знакомый с Linux, поехал собирать систему под AIX. Однако прошла неделя и от него приходили только редкие послания с описаниями новых проблем. Всплыла и старая проблема с утечкой памяти. И автор поехал на помощь.

Первый день они начали с экспериментов. Сделали тестовое приложение, которое работало со сторонней библиотекой, и через пару часов убедились, что память расходуется именно в ней. Заодно выяснилось, что устойчивая ошибка, приводящая к краху ядра, была обусловлена настройками памяти на уровне операционной системы, эти параметры позже исправил консультант. Еще несколько часов ушло на настройку альтернативного компилятора GNU C++.

Другая используемая библиотека просто не имела официальной версии под AIX для альтернативного компилятора, в наличии была только версия, сделанная «репрессированным» ими IBM-овским компилятором. В связи с долгим исправлением этих и прочих недочетов автору и его товарищу удалось освободиться очень поздно, когда поезда уже не ходили, а вызвать такси возможностей не было. Кое-как добравшись домой, они решили немного отдохнуть и приехать на следующий день позже.

На другой день опять не все в порядке. Начальство забыло предупредить IBM-овцев о том, что разработчики еще собираются работать. Но это утряслось и разработчики отработали еще один день, не завершив работу с одной серьезной ошибкой в модуле.

В третий день программистам удалось выяснить причину ошибки в модуле. Разработчики отправили основателю результаты и приступили к записи на диск наработанных за последние дни файлов. Приключение закончилось.

- **Хорошо там, где нас нет**

Читатель может подумать, что пример конторы автора непоказателен. Однако, по мнению автора, с подобными проблемами сталкиваются все.

- **Дефрагментация мозгов**

Современное софтостроение заслуженно забыто наукой.

Ошибочно полагать, что такая ситуация возникла недавно. Отрицательная динамика степени полезности технических книг наблюдалась еще в начале 90-х годов.

Нынешняя «гуглизация» позволяет быстро находить недостающие фрагментарные знания, зачастую забываемые уже на следующий день, если не час. Поэтому ценность книг как систематизированного источника информации, казалось бы, должна только возрастать.

Но для возрастания значения книг на фоне всеобщей фрагментации знаний необходимо умение аудитории читать и усваивать длинные тексты, неуклонно снижающееся последние десятилетия.

Является ли фрагментарное мышление приспосабливанием к многократно увеличившемуся потоку информации? Отчасти да, только это скорее не адаптация, а инстинктивная защита. Чтобы осознанно фильтровать информацию о некоторой системе с минимальными рисками пропустить важные сведения, нужно иметь чётко сформированные представления о ней, её структуре и принципах функционирования.

В такой ситуации альтернативой ухода от информационных потоков и некачественного образования во фрагментарную реальность является самообразование на базе полезных книг. Потому что хорошая книга – самый эффективный способ дефрагментации мозгов.

- **Простые правила чтения**
    - Быть достаточно ленивым. Чтобы не делать лишнего, не ковыряться в мелочах;
    - Поменьше читать. Те, кто много читает, отвыкают самостоятельно мыслить.
    - Быть непоследовательными, чтобы, не упуская цели, интересоваться и замечать лишь побочные советы.

Технические книжки – это не бессмертные произведения графа Толстого. К ним должен быть суровый, сугубо индивидуальный и безжалостный подход без какой-либо оглядки на чужое мнение. Прежде чем начать читать, возьмите листок бумаги, оптимально формата А4 – сложенный пополам, он удачно вкладывается в книгу, служа одновременно и закладкой. И запаситесь карандашом. Пользователям устройств для чтения электронных книг, к коим я тоже отношусь, необходимо освоить функции вставки в текст своих заметок. Если «читалка» этого не позволяет, придётся всё- таки обратиться к бумаге или сменить устройство на более функциональное.

Если, пролистав с полсотни страниц, вы не сделаете ни одной пометки, то можете смело закрывать сие произведение.

- **Литература и программное обеспечение**

Автор рассказывает о некотором, по его мнению, сходством между литературой и ПО. Например разделение на «автора» и кодировщика, немало сходства в процессе, фреймворки в литературе как некоторая база и проч.

Ответы на вопросы

- **Наиболее полезные пункты, по моему мнению:**
    - Три дня в IBM
    - Дефрагментация мозгов
    - Простые правила чтения

Глава 2. Темы для эссе

Белый Антон

Хорошие практики написания кода/чистки кода.

Каков он необходимый стек технологий для современного программиста?

Возможен ли дальнейший прогресс облачных вычислений и чего стоит ожидать в будущем?

Использование паттернов проектирование на реальных проектах.

Использование UML в работе. Возможные альтернативы.

Как можно поддерживать чистоту кода и зачем это необходимо?

Лень — двигатель прогресса.

Борисевич Павел

О том, каким должно быть хорошее резюме начинающего программиста.

О том, почему не стоит выбирать маркетолога главой компании по разработке программного обеспечения.

О том, как багом программы могут воспользоваться злоумышленники.

Эффективность использования средств генерации кода в разработке программного обеспечения.

О том, какие браузеры не стоит использовать в корпоративной среде для веб-приложений.

О том, почему ООП так быстро стало популярным.

О том, какие есть известные антипаттерны в разработке учётных программ.

О том, какие проблемы могут возникать, при сильном разрастании базы данных.

Использование модульного тестирования в разработке программного обеспечения.

О том, как определять необходимые слои корпоративной информационной системы.

О том, какие задачи удобнее решать, используя конечно-автоматную модель.

О том, какие трудности возникают после обновления программного обеспечения.

Гетьман Святослав

Почему IT-индустрия вряд ли сможет отказаться от переквалифицированных рабочих?

Почему школьные кружки по программированию не так популярны, как курсы переквалификации?

Как я отношусь к программированию: искусство или хлеб насущный?

Безысходность современного программирования.

Чем чреваты свои фреймворки и в чём плюсы узкоспециализированных сред?

Веб-разработка: зло или благо?

Software development: спасибо, что живой.

Двойственность (разработок) компании Microsoft.

Веб-разработка без ООП и ORM: миссия невыполнима?

Что же значит продажа долей Гейтса и Балмера?

Солдаты и генералы софтостроения: армейская и / или рабовладельческая иерархия?

Софтверные фирмы производят услуги?

Управление кейсами в ИТ.

Thinking in patterns, или приманка для неспециалистов.

Маркетологи в IT сфере.

Лучше плохой, чем никакой?

Всегда ли необходимо скрывать проблемы, возникающие во время разработки?

Как помочь it-сфере двигаться в полезном направлении?

Автоматизация глазами живого человека.

Какой должна быть система образования сейчас?

Григорьев Антон

Что выбрать: трёхмесячные курсы или четырёхлетнее образование?

Картина мира в голове человека.

Является ли софтостроение, программирование и т.д. искусством. Или суть и различия мотивации и стимуляции.

На тему парадигм программирования и подхода к написанию программ.

На тему создания языка программирования, сконцентрировав внимание на том, что должно лежать в основе языка (математическая модель и так далее).

Сравнительное про SQL и noSQL базы данных, отличия и преимущества.

Облачные вычисления.

Развитие технологий с коммерческой точки зрения.

Насыщенные интернет-приложения.

Потребности человека и их реализация в современном обществе.

Целесообразность использования шаблонов проектирования.

Эволюция профессии разработчика (опасность её упрощения).

Явления и сущности.

Правда ли мы что-нибудь понимаем?

Зависимость программных технологий от аппаратных.

Что значит быть специалистом?

«Гибкость» крупных компаний.

Пределы развития человечества.

Ипатов Алексей

Современный читатель и литература.

Влияние “гуглизации” на современное общество.

Технология программной фабрики.

Пакет SVN.

Гибкая (agile) разработка. Как это работает?

Компетентность специалиста в процессе должна быть не ниже архитектора соответствующей подсистемы.

Почему шаблоны «вдруг» стали ненужными? Нужны ли они вообще?

Можно ли отказаться от термина наследование?

Можно ли вообще считать или целиком причислить программирование к искусству, к ремеслу или к науке?

UML в наше время. Используется ли по сейчас?

Таблица остатков. Её основные назначения.

Использование программной фабрики.

Необходимость использования ООП и дальнейшая его судьба.

Карманные монстры и браузеры.

Новое пришествие CORBA в виде облегчённой её версии.

ЭВМ вымерли или вымирают.

О чём стоит призадуматься особенно желающим начать новый проект?

Лебедев Николай

Составление резюме.

Роль женщины в программировании.

Фреймворки и библиотеки.

Развитие браузеров.

Упадок IBM.

Как бродил Microsoft в веб-технологиях.

ORM фреймворки.

Безопасность запросов в БД.

CORBA. Подробнее.

Децентрализация ресурсов. Курс взят.

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

Типы OutOfMemory в Java и как их ибежать.

Как донести технологию до конечного пользователя?

Зависимость опыта от возраста.

Михальцова Анна

Влияние компьютера на жизнь человека.

Какие успехи в технологиях можно спрогнозировать через 20 лет?

Кто же всё-таки такой программист и в чём заключается его работа?

Программирование как искусство.

Стоит ли развиваться то статуса "гуру" в программировании, постоянно бегло изучать что-то новое, чтобы повысить спрос у работодателей, или всё же лучше оставаться "середняком", зато углублять свои знания строго в отведенной области?

Как при собеседовании для приёма на работу избежать, того, чтобы отбирать сотрудников не по "заучиванию тестов", а действительно по способностям?

Что наиболее важно в резюме: личные качества или навыки по специальности?

Что может быть мотивацией для программистов?

Доступность информации без использования технологий.

А безысходность ли безысходное программирование?

Технологии, использующиеся в веб-приложениях.

SQL. Перспективы развития.

Минусы развивающихся технологий.

Тонкий клиент.

Коробочный программный продукт. Понятие, использование.

Влияние имени на развитие компании.

Паттерны проектирования.

Зачем нужна история операций?

Отрицательное влияние синхронизации.

Повышение продуктивности «двусторонними» инструментами (two-way tools).

Развитие и актуальность БД.

А можно ли утверждать, что все методологии подобны между собой и отличаются слегка этапами реализации, их очередностью и длительностью?

Считается ли человек в сфере BI программистом?

Книжки по программированию для обсуждения.

Кто не ошибается, тот не пишет хороший код.

Ровдо Дарья

Проблема большого количества программистов и их компетентность.

Универсалы» и разработчики, владеющие специализированными технологиями.

Проблема некорректной работы приложений в разных версиях браузера.

SQL vs собственный язык запросов ORM-фреймворков.

Будущее облачных сервисов.

Современные команды 10-15 человек, вооружённые умопомрачительными средствами рефакторинга и организованными процедурами гибкой разработки, за год не могли родить работоспособный заказной проект, решающий несколько специфичных для предприятия задач.

Причины сложившейся ситуации.

Обычные художники заимствуют – великие воруют.

«Если они купили продукт – значит, он делает то, что они хотят. Если чего-то не делает – они все равно уже нашли workaround, им всё равно».

Junior developer и грязная работа.

Фрагментарное мышление.

Two-way tools.

Трубач Геннадий

Системы, которые генерируют код из какого-либо мета-кода.

Подход MSF

SQL, реляционные БД.

Проведение собеседования и анализ резюме.

Написание кроссплатформенных web-приложений.

Объектно-ориентированное программирование (его развитие, почему нужно использовать и что это дает).

Почему OS Windows намного лучше OS Linux.

Синхронизация транзакций.

Щавровский Святослав

Почему MVC столь популярен?

Захватит ли веб мир.

Будущее «невеб» приложений.

Ошибки Microsoft.

Есть ли будущее у SaaS.

Почему ООП провалилось?

Модульный подход к разработке ПО.

Размышления на тему будущего индустрии.

Паттерны проектирования - необходимость?

Тестирование - пациент жив или мертв?

Различия кодера и программиста.

Ярошевич Яна

Небезынтересной показалась мне тема о безвозмездной любви к своему делу, можно порассуждать на тему, насколько это верно в наших реалиях.

Также заманчивой показалась тема о понятии проявления творчества при написании софта. Можно поразмыслить на тему, является ли программирование творческим процессом.

Интересным мнением показалось мнение о важности коммуникабельности сотрудников, работающих в IT-сфере. Думаю, на этой почве может разгореться не один спор.

Последним интересным замечанием стало замечание о прохождении собеседования в IT-компанию на должность, близкую к разработчику. Действительно, обычные вопросы по технологиям, на мой взгляд, не помогут окончательно определить трудоспособность собеседуемого. Тогда возникает вопрос, как стоит проводить собеседование, чтобы добиться наилучшего результата.

«Гуглизация» как помеха получения систематизированных знаний.

Трата здоровья программистов во время дедлайна, как с этим бороться?

Чтение технической литературы: как оптимизировать этот процесс

Тестирование тестов. Важность тестирования. Сколько стоит уделять этому времени?

Важность средств автоматизации.

Как выбрать лучшую методологию для конкретного проекта?

Важность документации ПО.

UML и птолемеевские системы.

Плюсы и минусы UML.

«Обычные художники заимствуют – великие воруют».

Учитесь у лидеров рынка, их успех не от «просто так».

Изобретение велосипедов – пустое дело?

«Мыслить шаблонно» - где грань?

“Очень просто делать сложно, но очень сложно сделать просто”.

“Кто не хочет – ищет причины, кто хочет – средства”.

Можно задуматься на тему, является ли проект продуктом. Я бы попробовала доказать, что проект – это процесс, состоящий из многих стадий, в то время как продукт – это просто результат этого процесса. Аналогично нельзя, на мой взгляд, говорить, что проект и продукт – одно и то же. Возможно в жизни можно столкнуться с примерами, когда все эти три вещи сводятся к одному, однако в IT-сфере разница между этими понятиями существует.

Когда следует избегать ООП?

«Настоящий исследователь должен поменьше читать, чтобы иметь возможность больше думать головой». Тяжелый выбор между затраченным временем на обучение и качеством конечного продукта – как выбрать золотую середину?

Эти три пункта показались мне в большей степени информативными, что как-то не подталкивает на написание эссе. Однако, я могла бы поспорить на тему веб-приложений, как уже написала в предыдущем пункте.

Идея складывать программу из компонент, как дом из кирпичиков, переходя от одной целостной конструкции к другой

Написать самому или взять готовое, рискнув качеством?

Объяснение уменьшения количества женщин в IT-сфере с 1990-х годов.

Глава 3. Отчеты

Отчет 1 (Щавровский Святослав)

Общее впечатление

Объем материалов по первой части конспектов книги «Дефрагментация мозга» был большой. Но несколько огорчило, что в основном весь материал состоял из сухих конспектов, без большого количества своих мыслей. Причину тому можно описать одним веским, исчерпывающим выражением: «Первый блин комом».
Верю, что дальше будет лучше. И так, с надеждой, приступаю к отчету.

Основной лейтмотив

Основным лейтмотивом большинства конспектов было то, что «*не все так радужно и романтично, как может казаться многим молодым специалистам. Но не стоит принимать этот текст за подробную аналитику проблемы и чёткий план к действию. Скорее это побуждение к размышлениям и изысканиям».* (Григорьев А.)

Некоторая часть аудитории была не согласна с автором по поводу специализации, а именно с квалификацией инженеров:

> «Большая роль отводится инженеру. Всё-таки это не совсем программист. Возможно пару лет назад, в эпоху ЭВМ, так оно и было. Но сейчас инженер, в моем понимании, человек, имеющий больше отношения к конструированию, проектированию каких-то деталей, оборудования». (Михальцова А.)

> «В Беларуси, насколько мне известно, ситуация обстоит ровно также, как написано в источнике по поводу «прошлого», а именно в трудовых книжках в настоящее время до сих пор пишут «инженер-программист», добавляя «ведущий» или категорию для специалистов с высшим образованием, «техник-программист» для специалистов со средне-специальным образованием либо неоконченным высшим образованием». (Ярошевич Я.)

Большой резонанс вызвал раздел «О красоте». Примерно половина всех конспектирующих так или иначе написали в темах для эссе что-либо, связанное с красотой кода и софтостроения в целом. Об этом позже, а пока:

> «*Будет ли человек творчески подходить к написанию софта, не зависит от уровня образования, от количества знаний, приобретенных в учебном заведении, - зависит лишь от того, является ли он «творческой натурой». Один сделает красиво, но недостаточно функционально, а второй наоборот».* (Ярошевич Я.)

> *«Определение «техническое творчество» подходит больше. Это ни ремесло, ни искусство, ни наука, но, как говорится «aurea mediocritas», поэтому к делу стоит подходить творчески.» (Лебедев Н.)*

У некоторых удивление вызвало изречение, насчет безвозмездной любви к своей работе.

> *«В этом пункте описана идея безвозмездной любви к своему делу, что заставило задуматься». (Ярошевич Я.)*

Эта тема фигурировала в темах для эссе. Не было ни одного несогласного с тем, что «Хорошо оплачиваемая работа с творческим подходом к труду в современном мире – это привилегия, за которую придётся бороться всю жизнь».

Далее, переходя к найму на работу, я наткнулся на противоречие. Одни утверждали, что автор не прав:

> *«Большего всего, пожалуй, я не согласна со следующим высказыванием:*
>
> *«Не будет иметь большого значения то, что ты можешь сделать хороший дизайн, если за тобой на интервью придёт дилетант, заучивший десять известных работодателю «паттернов», 200 классов фреймворка и просящий за это в 2 раза меньше денег». Возможно, я слишком оптимистична, но я думаю, что много компаний сейчас на собеседовании спрашивают именно про те задачи, которые вы должны уметь решать. Например, если уже речь о дизайне, почти наверняка спросят, как лучше сделать такую или такую разметку. По такого рода вопросам и определяется ваш настоящий уровень, это важнее чем то, сколько атрибутов вы заучили. Я уверена, что работодатель думает так же». (Ровдо Д.)*

Другие им оппонировали:

> «Нынешние тесты на техническую часть не позволяют толком проверить реальные способности испытуемого, а лишь проверяют знания синтаксиса или какую-то странную логику искать правильный ответ в запутанных дебрях непонятно кем написанного кода, после прочтения которого хочется выключить компьютер и не включать никогда. Не самый лучший вариант, но что поделать». (Белый А.)

Коснувшись части, посвященной резюме, я встретил одно единственное несогласие с автором. Несмотря на свою единственность, такое мнение тоже имеет право на существование:

> «Как мне кажется, не всегда следует соблюдать краткость в своём резюме, если ты начинающий программист. Нужно максимально понятно объяснить с чем ты уже сталкивался и какие задачи ты можешь решать, чтобы потом не возникло недопонимание. Когда у программиста появится достаточно большой опыт работы, тогда, наверно, ему уже не обязательно стараться подробно рассказывать о всех своих приобретённых навыках, с опытом он будет понимать, что необходимо указать в резюме, а какая информация будет лишней». (Борисевич П.)

Темы для эссе

Самой популярной темой для эссе было, если обобщенно, «Является ли программирование творческим процессом». Темой, затронувшей умы аудитории несколько меньше, была тема, обобщенно названная «Резюме». Здесь я с аудиторией полностью согласен. Темы интересные и очень объемные. Интересной темой, встретившейся дважды среди предложенных, была тема «Как стоит проводить собеседование, чтобы добиться наилучшего результата». Также интересными темами мне показались: «Гуру программирования», «О безвозмездной любви к своему делу», «Проблема большого количества программистов и их компетентности».

Заключение

В целом хотелось бы заметить, что, для первого раза, аудитория поработала хорошо. Были интересные изречения, были интересные темы для эссе. Но не было достаточно много материала, чтобы было из чего выбирать. Надеюсь, что последующие темы будут лучше.

Отчет 2 (Гетьман Святослав)

1. Общее мнение

Компонентная сборка

Собирание «кубиков» выросло в большой рынок. Сама идея о том, что можно было бы делать программы, как аппаратуру, то есть собирать из компонентов, разумеется, полностью не осуществима, ведь предугадать все возможные потребности попросту невозможно, однако программисты стараются писать многократно используемый код. Будем считать, что это и будут компоненты в будущих программах. А в силу того, что опять же, всё предугадать невозможно, собрав из компонент программу, ограничиваются выборочным тестированием, полагаясь на вероятность.

Полное тестирование

Еще одна тема — полное тестирование всех возможных исходов, которое открывает глаза на то, насколько запутанным может быть путь от начала программы до ее завершения и сколько разных вариаций входных и выходных данных может принимать и отдавать программа.

Можно оценить количество состояний, которые необходимо охватить для полноты модульного тестирования при конвертации валют: ISO 4217 даёт список из 164 валют. Предположим, что наши входные данные:

- имеют только два знака после запятой;
- значения положительные;
- максимальная величина – 1 миллион;
- дата конвертации всегда текущая;
- мы используем только 10 валют из 164.

Итого: $10^2 \times 100\,000\,000 = 10\,000\,000\,000$.

На практике же программист применит эвристику и будет тестировать, например, только несколько значений (один миллион, ноль, случайная величина из диапазона) для нескольких типовых конвертаций из 100 возможных, дополнительно проверяя допустимую точность значений на входе. При этом формальный показатель покрытия модульными тестами по-прежнему будет 100 %.

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

Конечно-автоматная модель

Конечно-автоматная модель, как мне кажется, не самая удобная, и пользоваться ей можно только для решения небольшого класса задач, при этом время отклика компонента оценивается приближённо, а тестирование в большинстве случаев происходит выборочно.

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.

Женщины в IT[1]

Автор отмечает, что количество работающих в софтостроении женщин росло до 1990-ых годов, после чего резко пошло на убыль. Автор утверждает, что это связано с начало массового использования принципов ООП.

С тем, что ООП сыграла не последнюю роль в вытеснении женского труда из отрасли IT, я не соглашусь. По моему мнению, объектно-ориентированный подход ну никак не мог этому поспособствовать.

Не считаю, что переход к ООП являлся сложным вообще для любого пола, просто взгляд на вещи в программировании был немного изменен. Не думаю, что женщины не справились с этой задачей, поэтому их количество начало падать. На мой взгляд, это могло произойти с **ростом престижности работы программиста** и **количеством зарабатываемых средств**. В связи с этим, мужчины просто могли начать «вытеснять» женщин с этого рынка, так как им свойственно стремиться к такой профессии больше в этой ситуации.

Кое-кто высказал мнение о том, что такая проблема связана с неумением разбираться в технологиях софтостроения. «Впоследствии мне не раз приходилось видеть исходники программисток на вполне себе объектно-ориентированном Delphi/C++Builder» - честно говоря, немного обидно это читать. Уверена, что на этапе перехода к этим языкам программирования, у программистов мужского пола исходники тоже недалеко ушли. Опять же, если говорить о настоящем моменте времени, программисток не так уж и мало. В довесок можно сказать, что наоборот большинство компаний хвастаются, что стали нанимать на работу больше женщин, афроамериканцев и латинос.

Снижение количества женщин в отрасли из-за «резко» возросшей популярности универсальных сред – лишь одна сторона медали, затронутая автором. На мой взгляд, женский пол с лёгкостью способен разбираться в сложных математических абстракциях и работать в более универсальном пространстве, недели в коридоре узкоспециализированных программных сред. И ссылаться на половую принадлежность, как это делает автор, грубо.

Почему же тогда девушек меньше? Увы, к сожалению, далеко не каждую девушку в школе и дома обучают так, чтобы к университету у неё было не просто образно-гуманитарное представление мира, но и формализовано-техническое, которое позже позволит ей стать специалистом в IT-сфере и без труда справляться с теми проблемами, с которым справляются мужчины.

Глупый / умный заказчик

Хочется по своей наивности верить, что далеко не все заказчики представляют собой «большой глупый кошелёк», как это с одной стороны было продемонстрировано в главе «Диалог о производительности». Уверен, что есть компании, которые «разрабатывают сами на себя» (к примеру, EPAM и гиганты вроде Google, Microsoft, Apple), где заказчиком выступает человек из компании (CEO).

Фреймворки и безысходное программирование

Когда вы имеете дело с черным ящиком, вы до конца не понимаете, как он работает. Без исходного кода возникают трудности в понимании внутренней логике работы программы. И тестирование

[1] Примечание эксперта: Половина группы не согласна с мнением автора. Стоит отметить, что все девушки высказались по этому поводу.

такого кода во многих случаях просто бессмысленно, потому что нельзя с уверенностью предугадать реакцию системы.

В конечном итоге, это может привести к безысходности и безнадеге. А отчего безысходность и безнадёга? Да оттого, что слепое использование фреймворков и библиотек ведет к непониманию процесса работы программы и к уменьшению ее производительности.

На мой взгляд, надо чётко понимать, что где-нибудь в NASA или на военных разработках, они используют узкоспециализированный софт, чтобы проводить более точные расчеты. Не исключен также и вариант написания той же NASA своего программного решения (если хотите, то это будет фреймворк), которое позволит им сделать производительность повыше и точность побольше (зависит от исходной цели разработки данного продукта). Затем этим фреймворком могут воспользоваться другие компании, которым необходим аналогичный функционал и экономия времени с деньгами на разработку. Плохого в том, чтобы не тратить время на «создание велосипеда», я не вижу (только если это не университет, где написание фреймворка даёт пишущему чёткое анатомическое представление того инструмента и последующих, которые он будет использовать).

«Частным случаем безысходного программирования является софтостроение без использования системы управления исходным кодом» - я так понимаю, что тут слово «безысходное» имело свое прямое значение, а не заявленное ранее в данном пункте. Иначе я не просто не согласна с этим утверждением, а вообще его не понимаю. На дворе 2015-й год, и мной одолевают очень сильные сомнения, что за пределами университетов есть курсы и / или предприятия, которые пишут рабочий и окупаемый продукт, не используя Git или другие ревизионные системы.

К сожалению, многие сейчас забывают о «чёрных ящиках», ошибочно полагаясь на свою сообразительность, либо просто не понимают, откуда ноги растут. Однако я, в процессе разработки, всегда лезу «под капот», пытаясь понять, как эта штука работает. И всем советую!

ООП

Не согласен с тем, что «ООП реализовано повсеместно без малейших представлений о его применимости». ООП активно используется в нынешнее время. ООП принесло в программы такой важный момент, как безопасность, возможность наследования, полиморфизма.

Эволюция аппратуры и скорость разработки

С приходом глобальных поисковых сервисов, ценность информации снизилась. Притом существенно. Теперь ценно не столько владение информацией, сколько возможность получить её за ограниченный срок времени; когда нужная информация всегда под рукой в большом объеме, необходимо уметь отыскать в этом объеме необходимое.

Хорошо это или плохо - сложный вопрос, достойный отдельного эссе. В связи с такой доступностью информации, технологии получили новый толчок. Нельзя сказать, что толчок в нужную сторону. Возможно, случилось небольшое отклонение или виляние, но определенно толчок случился. Но осталось неизменным то, что технологии - основа нашей индустрии.

С увеличением количества сервисов, функциональности которых слегка (или не слегка) пересекаются, появилось совершенно понятное желание программистов сократить себе работу, а именно разработать новый «модульный» подход к разработке ПО. Модульность уже давно и легко реализуется в аппаратном конструировании. Это обусловлено жесткими входными и выходными данными последнего. В программном конструировании такого «совершенства» добиться сложно. Однако попытки были предприняты и предпринимаются до сих пор, притом с успехом. Хотя насчет «успеха» можно поспорить. Давно известно, что нет таблетки от всех болезней. Опираясь на этот популярный факт, можно с ответственностью заявить: подлинного модульного подхода к программированию ПО быть не может. Может быть только нечто похожее. Но отсутствие таблетки

от всех болезней, не исключает наличия избавления от всех болезней - смерть. Я не думаю, что это может постигнуть нашу индустрию в целом, но некоторых частей определенно коснется.

Автор утверждает, что по сравнению с 80-ми годами скорость разработки снизилась. На мой взгляд, его доводов отнюдь не достаточно. По сравнению с 80-ми годами сейчас в программировании ставятся абсолютно другие задачи, более трудоемкие и требующие большего внимания и времени, так как мы работаем сейчас с абсолютно другими технологиями. Поэтому считаю, что сравнивать скорость разработки некорректно в этой ситуации.

«Вчерашний студент теперь пишет не на ассемблере и C, а на Java, будучи уверенным в принципиальной новизне ситуации, не всегда осознавая, что изменилось только количество герц тактовой частоты и байтов запоминающих устройств», «и хотя по срокам, бюджету и количеству разработчиков владеющие специализированными технологиями выигрывают у бригады «универсалов», но возникающие при этом риски могут свести к минимуму весь выигрыш». Во-первых, начинает складываться впечатление, что автор книги — пессимист. Во-вторых, на мой взгляд, на данный момент как раз ситуация, в которой «универсалы» ценнее, сменяется той, в которой если ты хочешь быть более востребованным, стоит углубить свои знания в некоторой определенной узкой области.

За время эволюции, аппаратная среда росла в производительности. В некоторых аспектах она уже упирается в потолок, или близка к нему. Процесс производства аппаратных продуктов наладился, подешевел, стал более качественным. Производительность «железа» возросла на порядки. Стоимость тоже на порядки, но снизилась. Увеличилась надёжность, развилась инфраструктура, особенно сетевая. Параллелизация вычислений пошла в массы на плечах многоядерных процессоров.

Не это ли называется эволюцией?

Программная среда же как опиралась на принципы середины прошлого века, так и опирается. Да, появилось множество новых языков программирования, новых подходов, новых шаблонов проектирования. Однако смысл остался тем же. Так как 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.

Не берусь оценивать данный факт. Вполне возможно, что это вовсе и не плохо. Может еще не пришло время?

Заключительное слово

Поднятые темы про эффективность разработки, про профессионализм в плане применимости технологий и идей, сравнение подходов к разработке софта и аппаратуры – всё это безусловно занятные темы, и они заставляют поразмыслить о своей компетентности, о своём профессионализме. То есть мало того, что такие источники являются пищей для размышлений, они в довесок могут стать своего рода стимулом к росту, к развитию.

2. Мнение эксперта

Самые часто затронутые темы:

Снижение количества женщин в IT-сфере

Затронуло минимум 6 человек, причём трое из них – это женская часть нашей группы, а именно: Анна Михальцова, Дарья Ровдо и Яна Ярошевич. К ним высказать мнение присоединились Святослав Гетьман, Алексей Ипатов, Геннадий Трубач.

Эволюция аппаратуры и скорость разработки

Вторая по популярности тема, на которую тоже высказалось 6 человек: Антон Григорьев, Николай Лебедев, Геннадий Трубач, Святослав Щавровский и Яна Ярошевич.

Безысходное программирование

4 человека: Алексей Ипатов, Анна Михальцова, Дарья Ровдо, Яна Ярошевич

Конструирование программ как аппаратуры

3 человека: Антон Белый, Дарья Ровдо, Яна Ярошевич

Фреймворки vs узкоспециализированные среды

В явном виде у двух человек (Гетьман, Лебедев), а вообще, тема тоже оказалась довольно популярной.

Самые интересные / популярные темы для эссе:

Тенденции в разработке / будущее индустрии
Конечные автоматы
Сборка программ как домика из кубиков
ООП
Роль женщины в программировании
Фреймворки и биилиотеки
Безысходность в программировании
Гуглизация / доступность информации
ЭВМ нового поколения

На мой взгляд, стоило бы написать эссе на темы:

Какие задачи удобнее всего решать, используя конечно-автоматную модель (Павел Борисевич)
Модульный подход к разработке ПО (Святослав Щавровский)
ЭВМ нового поколения (Алексей Ипатов)

Отчет 3 (Лебедев Николай)

1. Общее мнение

Данная часть книги имеет, в большей степени, ознакомительный и информационный характер. Здесь автор рассказывает про смену направления в разработке и про технологии, которые имели при этом место. В добавок ко всему, мы не имеем большого опыта в реальной разработке и не «прочувствовали» на себе этот переход. Именно поэтому свое мнение высказать было трудно, а дать одобрительную/негативную оценку еще труднее. Однако, кое-какие мысли можно высказать…

Первое, что встречается при прочтении - «Карманные монстры». Не совсем понятно, о чём хотел сказать автор в этой главе. Разработка неоправданно усложняется за счёт недостаточной гибкости и квалифицированности разработчиков и менеджеров? Скорее всего, разработка проекта если и усложняется (по настоянию так сказать менеджера), то только ради повышения логичности и структурности проекта и кода. То есть, всегда очень легко быстро написать код как ты его понимаешь и как он сразу пришёл к тебе в голову. Так называемый метод «Лишь бы работало». Но всегда стоит задумываться о качестве своего кода, об его эффективности. А сколько раз и достаточно опытным программистам приходится "копаться" в абсолютно новом для них материале. Как будто "карманные монстры" страшны только начинающим. С этой проблемой сталкиваются и опытные разработчики, а не только новички.

Сложилось впечатление, что веб-разработка автором едко задвигается в далёкий тёмный угол, дабы не мешала разрабатывать и развёртывать настоящие полноценные софтины для декстопников. Автор явно намекает, что более старые технологии выигрывают у новых в сложности написания кода, в его количестве. Как будто ставятся под сомнения плюсы этих самых новых технологий. Может оно и так, но вряд ли приложение на Delphi сейчас может тягаться с симпатичными и функциональными веб-приложениями. Автор крайне однобоко смотрит на возможности веб-приложений, а именно на работу со встраиваемыми скриптами. На его взгляд, работа с такой технологией чрезмерно трудоемка. Хотя и опасения автора тоже понятны, но, скорее, с точки менеджера: времени действительно стало уходить больше на всякую «мишуру» и «конфетти», особенно это видно в чётком разграничении должностей и сфер, типа «разраб» и «дезигнер».

Было интересно узнать про историю веб-сферы, а в частности о веб-браузерах. Сама идея о создании универсального, программируемого терминала, которым является веб-браузер, поддерживающий стандарты взаимодействия с веб-сервером, показалась очень интересной. Непонятно только, почему ничего не было сказано о их классификации и развитии JavaScript на этом фоне.

Как ни крути, но, увы, нельзя вот так просто взять и забыть о сфере веб-разработки даже на микросекундочку, ибо рынок уже как с 2008 года дядюшкой Стивом Джобсом

построен и заточен под мобильные технологии, которые направлены на максимальное удобство и (что вполне логично, исходя из названия) мобильность. Так почему автор описал Java как совсем провалившеюся технологию? Снова мы не можем ответить, а может просто не поняли автора.

В утешение будет сказано, что софт для десктопов и суперкомпьютеров до сих пор спонсируется и разрабатывается, изменилось лишь отношение к нему, как, впрочем, и к стационарным ПК с ноутбуками.

Имеют ли сейчас современные технологии проблемы? Да, конечно. Очень многие из них автор четко описал. Что же будет дальше? Время покажет.

2. Мнение эксперта

Условно всех высказавшихся можно разбить на две группы. Кто-то был не согласен с автором в некоторых моментах, а кому-то было трудно говорить о чем-либо в повествовательной части книги. Кто-то занимал позицию в центре этих двух направлений. В любом случае, у каждого было свое мнение, и оно повлияло на итоговое общее мнение. Естественно, что-то было добавлено от меня самого, чтобы как-то отобразить общий смысл.

Что было инвариантно у всех? Абсолютно все защищали современные технологии, которые мы изучаем сейчас, однако большинство признает, что и они не без изъяна. Оно и понятно. Это же логично, современное => инновационное => лучше, но в современном мире, который живет по законам маркетинга и конкуренции, не всегда это так. С этим также многие согласились.

Как следствие, у всех вызвала большой интерес история веб-сферы, а, в частности, веб-браузеров. Практически все указали эту тему для эссе или близкую к ней. Только Гетьман Святослав и Лебедев Николай, т.е. я, указали другие темы. У остальных мнения совпали. Поразительно, на мой взгляд, но, вместе с тем, нет разнообразия для выбора. С связи с этим, придется согласится с большинством и выбрать тему **«браузеры, которые не стоит использовать в корпоративной среде для веб-приложений»**, автором которой является Борисевич Павел. Я ее посчитал самой интересной из предложенных тем о браузерах.

Апплеты, Flash и Silverlight – самый популярный радел. Большинство указало его, как самый интересный. Тут большая заслуга автора.

Как оказалось, анализировать общее мнение очень интересно, хотя и долго. Интересно это тем, что, после объединения, на выходе получается очень интересная

статья. Как у эксперта у меня есть возможность подвести какой-нибудь итог всей нашей работы по этой части книги. Ну и итог на сегодня вот такой.

Понятное дело, что и в современных технологиях есть большие изъяны, которые четко были описаны автором. Почему так происходит? Причин очень много. Некоторые из них нам были изложены. Возможно, не все аргументы были убедительными, но абсолютно все признают наличие этих проблем. Однако есть более важный вопрос: чему же все-таки научила нас эта часть книги? Ответить на него можно так: в любой технологии нужно замечать как достоинства, так и недостатки, чтобы потом грамотно оперировать ими.

Отчет 4 (Щавровский Святослав)

Общее впечатление

Объем материалов по четвертой части конспектов книги «Дефрагментация мозга» оказалось много. Очень порадовало, что, в отличие от первой части, здесь оказалось много мыслей писавших. Это в очередной раз показывает, что «первый блин комом». Приступаю к отчету.

Основной лейтмотив

Почти все конспекты в том или ином роде содержали изречения о мнении автора по поводу ООП и, так названного им «Ада Паттернов», притом как отрицательные:

> *«Автор гнет очень жесткую линию относительно ООП. Он очень часто упоминает проблемы ООП. Но не нужно забывать, что каждая парадигма имеет свои плюсы и минусы. ООП является самым популярным подходом в настоящее время». (Трубач Геннадий)*

> *«Дааа, конечно же, поварами, заварившими это блюдо, состоящее из спагетти-кода и фрикаделек неумения (недостатка практики) и заправленное соусом сжатых сроков, являются именно ООП и «Ад паттернов», а никак не программисты, несомненно. Если же привести в пример какое-либо API современной мобильной операционной системы, то вы увидите: нет ничего более красивого и стройного, чем логика этого самого API. Я не берусь разбирать Android API по паттернам, но уверен, что с ними полный порядок». (Щавровский Святослав)*

так и положительные:

> *«Взгляд на ООП, описанный в книге, я полностью поддерживаю. То есть как, полностью. Я уверена, что ООП, конечно, вещь полезная, в некоторых случаях работать с ней приятно, ощущаешь удовлетворение от того, что решение проблемы такое элегантное. Однако зачастую оно оказывается и слишком сложным. Это огромное количество интерфейсов, эти наследования, иногда смотришь и думаешь, зачем все это, ради какой-то небольшой функции? Может, самой ее реализовать по-быстрому?» (Ровдо Дарья)*

> *«Если говорить об ООП, то в целом с мнением автора можно согласится. «Ад паттернов» и все из него выходящее действительно имеет место». (Лебедев Николай)*

> «До того, как я прочитал этот материал, я не видел никаких недостатков в ООП. Мне казалось, что ООП – это самое удобный и вообще правильный способ программирования. Раньше я не задумывался о тех проблемах, которые могут возникнуть, если использовать ООП в больших проектах». (Борисевич Павел)

> *«Касательно позиций насчёт ООП тоже согласен не полностью, ибо паттерны для того и созданы, чтобы облегчить построение более сложных систем. Да, правильно было замечено, что во многом, если не во всём, они представляют кальку с природы, перенесённую на связи и принципы ООП. Опять же, мы довольно ограничены в технологии (а может и фантазии), чтобы воплотить в жизнь мечту автора (и не только его): сделать связи красивее, элегантнее и настолько проще, чтобы можно было не запутываться в построении сложных систем». (Гетьман Святослав)*

Легко заметить, что положительные мнения преобладают над отрицательными, что для меня стало откровением.

Столь же обсуждаемым стало размышления на тему ORM. Опять же были как согласные, так и несогласные:

> «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». (Лебедев Николай)
>
> «По поводу ORM, мне нравится идея проецировать классы из ООП языков в базы данных без особых проблем. Это позволяет получить меньше проблем с эдаким конвертированием одного типа информации в другой. Но, безусловно, те, кто пользуется ORM технологиями, должны знать, как это все работает на более низком уровне». (Белый Антон)
>
> *«В принципе, все сказанное про ORM, по моему мнению, верно. ORM действительно упрощает жизнь разработчикам, но одновременно с этим, программисты не хотят разбираться в БД и языке SQL. Всегда нужно по правильному, сначала разобраться с БД, потом как с ней работать и что такое SQL, а только потом переходить к ORM, так как на деле будет допускаться огромное количество ошибок, некоторые из которых не вылезут сразу». (Трубач Геннадий)*

Помимо суждений/осуждений автора, были еще и очень интересные пассажи на эту(и не только) тему:

> *«Очень хороший пример был из жизни автора с функцией авторизации. SQL vs собственный язык запросов ORM-фреймворков. Актуально как никогда. Перед тем, как прочитать данную часть книги, я потратила 3 часа на написание сложного запроса, используя Criteria в Hibernate. 2,5 часа из них я мысленно ругалась. Этот же запрос я написала днем ранее в 3 строки на SQL, но интересно же было узнать, что там придумал Hibernate, чтобы было удобнее разработчикам. Итог — 30 строк кода. И еще при этом вообще непонятно, как реально выглядит запрос в базу данных. Подозреваю, что я буду еще раз писать с использованием Criteria только если меня очень сильно попросят». (Ровдо Дарья)*
>
> *«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. Используя 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». (Лебедев Николай)*
> *«Пока что никуда мы не денемся от понимания на уровне паттернов, однако, я считаю, что не за горами языки или системы, в которых реализация паттерны будет проводиться так, что нам не нужно даже знать особую структуру в зависимости от случая, просто написать пару инструкций». (Гетьман Святослав)*
> *«В книге проскакивает очень интересная мысль о том, что технология тем эффективней, чем более идеален моделируемый ею мир (и заметка о том, что ООП этому критерию как раз-таки не удовлетворяет). Но не понятно, что подразумевается под идеальным миром. Мир, наиболее приближенный к реальному? То есть, наименее абстрактный мир? Но почему такая технология будет более эффективной? Может потому, что с неабстрактными вещами*

банально легче работать (ведь мы и так каждый день «работает» с нашим неабстрактным миром). Или потому, что наш мир уже идеально спланирован, и зачем планировать свой, который может быть далеко неидеальным. То есть лучше отталкиваться от чего существующего и реально работающего, чем создавать что-то принципиально новое, рискуя что-то не учесть? Не знаю, на самом деле я не очень понял эту мысль». (Григорьев Антон)

Мнения оказались очень интересны и разнообразны.

Книга «Дефрагментация мозга» интересна тем, что создает большое количество тем для размышлений, несогласий и разглагольствований. Очередной темой стал рефакторинг кода:

> «Интересно, что автор много раз повторяет, что надо рефакторить код. Конечно, все понимают, что это очень важно, особенно на долгоиграющих проектах, однако многие попросту забывают/забивают. А затем сами на себя ругаются, когда пытаются добавить новый функционал, на основе уже имеющегося». (Белый Антон)

> «Когда-то на каком-то сайте я даже читала статью о том, что нужно делать разработчику, чтобы закрепить за собой своё рабочее место. Среди перечисленных пунктов был весьма интересный: вопреки всем правилам клинкода писать код так, чтобы его смог в дальнейшем разобрать только тот, кто его и написал. Этот пункт весьма отдаленно граничит с тем, что имел в виду автор книги». (Михальцова Анна)

Темы для эссе

В этот раз темы для эссе почти не пересекались. У каждого была своя, уникальная, тема, что не может не радовать. Вот некоторые, наиболее интересные, из них: «Каков он необходимый стек технологий для современного программиста?», «Веб-разработка без ООП и ORM: миссия невыполнима?», «Тяжелый выбор между затраченным временем на обучение и качеством конечного продукта – как выбрать золотую середину?», «SQL vs собственный язык запросов ORM-фреймворков», «Необходимость использования ООП и дальнейшая его судьба», «Почему ООП провалилось?».

Заключение

В этот, четвертый по счету, раз конспекты стали сильно лучше, чем в предыдущий (первый). И мнений стало больше, и сами мнения стали более объемными и четкими. Уже чувствуется, что книга заставляет обдумать прочитанное. В некоторых конспектах уже появились суждения на счет самого автора, что не может не радовать, потому как это означает, что книга зацепила:

> *«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». (Лебедев Николай)*
> *«Всё меньше и меньше начинает нравится подход автора к новым веяниям, начинает чувствоваться субъективная неспособность перестроиться. Конечно, это, с одной стороны, даёт массу тем для дискуссий, с другой стороны, формирует однополярное мнение, какую-то избитую категоричность, из-за которой проблема хоть и разобрана, но только с одного боку». (Гетьман Святослав)*

Этот факт не может не радовать. Стоит отдать должное автору. Подытоживая, скажу: конспекты стали больше и интереснее. Чувствуется, что это только начало.

Отчет 5 (Гетьман Святослав)

3. Общее мнение

Эта часть книги мне не сильно понравилась. (Белый)

Тут больше проводилось какое-то сравнение ***неизвестных*** мне технологий, часть из которых ***устарела***. (Белый)

«Как уберечь кукурузу от насекомых-вредителей? Очень просто: выкосить её всю, к чертям. Вредители придут, а кушать нечего.» (Трубач)

И снова критика, еще с первых страниц. (Трубач)
оформление книги ужасно (Трубач)

Тема облачных технологий

Облачное вычисление. Очень популярная и используемая технология. Можно сказать, что все, что связано с Интернетом, относится к ней. Ведь архитектура такова, что вся логика выполняется, в большинстве своем, удаленно на сервере, а нам приходит результат. Это стало ***совсем обычным делом***, ведь сегодня скоростной доступ в Интернет не является чем-то заоблачным. (Белый)

Полагаю, всё это [тема ВКЦП] относится к промышленному и корпоративному использованию. ***Вряд ли рядовому пользователю нужны такие средства***. (Григорьев)

Не понятно, к чему был рассказ о возросшей и неиспользуемой мощности личных терминалов. Как это относится к облачным вычисления, если речь идёт как раз-таки о перекладывании работы с личных терминалов на внешние сервера? Такое чувство, что ***автор просто не определился, о чём говорить в главе***. (Григорьев)

[ВКЦП = Вычислительный Центр Коллективного Пользования, советское определение] Для советской страны в 1972 году первым сделать такие прорывные попытки, при этом затронув важную связь между экономикой, государственной сетью и объединением региональных центров, было просто великолепным подспорьем, которое, глядишь, могло бы сделать много полезного шума и предоставить конечному пользователю (да и государству тоже, об этом не забывать) много удобств. (Гетьман)

Внедрение подобных систем было очень перспективно из-за централизации производства и административного аппарата. (Лебедев)

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. (Лебедев)

ВЦКП пролетело:

3) **Профуканная миниатюризация**
4) **Просчёты и развал страны**

Я бы ещё добавил в этот список самое по себе понятие ***«лихие 90-ые»,*** когда работать надо было за еду, когда во все сферы жизни стал вмешиваться рынок, отчего налаженная до того система «мороженное за 3 копейки» превратилась в чёрт знает что. Понятно, что рано или поздно в

программирование примешалась бы экономика, расширив и размыв понятие, но то, что это у нас произошло так бурно и стремительно (без подготовки, как на Западе) ещё раз, на мой взгляд, доказало, что ~~новые русские не сахар~~ стоит уделять время прогнозированию будущих решений и всегда держать ухо востро. (Гетьман)

Возврат к централизованным системам возможен. И я даже знаю почему.
Это предлагает тем же геймерам, а значит, одной из самых «жирных» прослоек современного инфообщества неописуемые возможности, равно как и прибыль таким корпорациям, как Sony, Nintendo, Microsoft. Если подробнее, то не секрет, что та же Sony ведёт разработки «игровых подписок» для конечного пользователя, что из себя представляет тот же централизованный подход, но уже по отношению к играм. Фактически, игрок будет играть в одну и множества доступных за определённую плату терминалом «игровых проекций», которые запущены / поддерживаются сервером-«облаком».
Плюс, эта система уже активно используется, вспомнить хотя бы банковское дело. Программисты в банковской сфере не только много получают за прогнозирование рынка, но и за создание таких централизованных систем и поддержку безопасности для терминалов, коими и являются небезызвестные банкоматы.
На мой взгляд, за этими подходами будущее. Но всегда стоит помнить, что даже у такой схемы есть скрытые слабые места как банальное падение серверов и, как следствие, пропадание всего функционала на терминалах. Как это обойти – время покажет. (Гетьман)

Прогресс технологий

Надо подходить к этому делу не только с финансовой точки зрения, но и думать о ***совместимости*** своих версий. (Белый)

Также автор как бы высмеивает ту частоту, с которой некоторые компании выпускают новые версии своих продуктов. Понравилась фраза: *«...можно перескочить со второй версии сразу на четвёртую, минуя третью и третью с половиной. Правда, уже анонсирована пятая*». (Белый)

[Вычислительная мощность ПК не используется на 100%.] Неудивительно, ибо рядовой пользователь выполняет с помощью ПК достаточно нехитрые задачи. В тех случаях, когда он выполняет что-то сложное, что нагружает ЦП и ОЗУ, то, во-первых, стоит задуматься, а ***является ли рядовым*** данный пользователь (может быть, это программист вроде Константина Антоновича Зубовича, которому захотелось «поиздеваться» над программой, используя её слабые места), и, во-вторых, делает это всё тот же рядовой пользователь неявно, с помощью вспомогательных средств. Это, в частности, относится к геймерам: они ведь не производят ручные расчёты на матрицах, не влезают в дебри линейной алгебры и теории операторов, чтобы получить ту картинку и тот функционал, который у них есть благодаря тому, что кто-то из числа оговоренных выше лиц (программистов и не только) уже со всем этим достаточно «накувыркался». (Гетьман)

[Из-за избытка мощностей на узлах (в случае распределённой системы) происходит переход от распределённых систем в пользу централизованных.] И я считаю, что это правильно, что если есть излишки в системе, которыми никто не пользуется (опять-таки, надо понимать, когда это «никто» не такое уж и пустое множество), то от них надо избавляться. (Гетьман)

Большие ЭВМ не вымирают и не вымрут никогда, ибо они являют собой образцы стратегичности, требовательности в купе с защищённостью и производительностью. А за это многие дядьки с тугими кошельками отдадут денюжки. (Гетьман)
Согласен, что большие ЭВМ (типа мейнфреймов) являются неотъемлемой частью сервера. Никакие ПК не смогут заменить мейнфреймы! (Трубач)

Очень понравилась мысль: **«Солдаты от софтостроения не должны рассуждать, ибо генералы договорились с поставщиком». Где тогда грань между рабским трудом и профессией программиста?!** (Гетьман)

SaaS

Хороший вопрос: «софтерные фирмы производят услуги»?

Соглашусь с точкой зрения автора: услуги не производят, а ОКАЗЫВАЮТ. А фирма производит ПРОДУКТ, а не услугу. Разница между ними в том, что продукт можно хранить, он физически осязаем, и время и место его потребления могут растягиваться, чего не скажешь об услугах: оказал, и она тут же «нашлась к месту».

Если уже принять тот факт, что фирмы = продукт, то без схем, описанных автором, не обойтись.

3. Производитель создал продукт, продукт обязательно проходит по рукам дистрибьюторов (консультанты, продавцы и т.д.) до конечного пользователя.
4. Программа как услуга сразу доходит до конечного пользователя, минуя дистрибьюторов (единственный это сам Интернет). Если пользователь программного продукта работает с ним через ВЦКП, то последний предоставляет доступ пользователям к налаженной системе.

Из чего следует, что в первом случае «навар» и проценты приведут к излишней дороговизне продукта, который может быть не до конца отлаженным, а во втором случае есть риск краха всей системы ВЦКП. Но и надо отметить, что рынок добрался и до второго пункта: появилась цифровая дистрибуция, люди за это получают денег не меньше, чем за обычную, «аналоговую». (Гетьман)

"Значит ли это, что софтостроительные фирмы теперь, как утверждают некоторые маркетологи, «производят услуги»?" Прочитав этот вопрос в книге и учитывая достаточно смелый характер изложения материала автором, боялась прочитать дальше положительный ответ. Но не тут-то было. Авторское "Разумеется, нет." весьма и весьма порадовало. Хочется всё-таки и дальше верить, что софтостроение останется "мощным двигателем" решения проблем, грубо говоря "производителем" основы для услуги, но не скатится до просто "услуги". (Михальцова)

Я бы попробовала доказать, что проект – это процесс, состоящий из многих стадий, в то время как продукт – это просто результат этого процесса. Аналогично нельзя, на мой взгляд, говорить, что проект и продукт – одно и то же. Возможно в жизни можно столкнуться с примерами, когда все эти три вещи сводятся к одному, однако в IT-сфере разница между этими понятиями существует. (Ярошевич)

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... (Лебедев)

CORBA

Sun могла объединиться с CORBA и никто бы из двоих не улетел, все были бы в выигрыше и концепцию довели до ума. (Гетьман)

SOA появилась как хорошее, но сырое подспорье CORBA и её сырость заключалась, по словам, автора, в вынужденных надстройках для достижения базового для CORBA уровня и простоты.

Зачем тогда необходимо было пересаживать всех и вся на сырую, но новую землю – я не могу понять. Возможно, было просто дешевле, а это, как на зло, не такое уж и программирование, чистый маркетинг, поднасоливший индустрии. Или, быть может, не всё так прозрачно? (Гетьман)

Обратив внимание на различные декларации и нелегкий путь бекенд разработчика в среде Java, трудно не согласится, что CORBA выглядит намного привлекательнее (Лебедев).

Новые технологии

Удивило заявление, что новые технологии штампуются (даже когда от этого один вред) лишь для заманивания потребителя, то есть в маркетинговых соображениях. Хотя, такое действительно можно заметить. Например, продукция компании Apple, которая уже давно высасывает «новые технологии» откуда попало и делает из этого революцию. И ведь народ ведётся. (Григорьев)

Совсем не всегда обновления являются полезными, нужными или просто понравившимися. (Ровдо)

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. (Лебедев)

"…отказаться от «новых» технологий рядовым разработчикам непросто, особенно работающим в сфере обслуживания под руководством менеджеров среднего звена с далёким от технического образованием, оперирующих понятиями освоения и расширения бюджета и массовости рынка специалистов, а не технологической эффективностью." Меня давно интересует вопрос, почему обычными простыми разработчиками командуют люди, далекие от технических понятий. Может проблема не в том, что "начальники" не могут организовать рабочий процесс, а в том, что не находятся такие люди, которые сделают это хорошо? (Михальцова)

"Экономика потребления обязана крутиться, даже если в ней перемалываются миллиардные бюджеты бесполезных трат на модернизацию, переделку и переобучение." Я считаю, что если запущен механизм "разработки прогресса", то стоит исключить такие слова, как "бесполезная трата". Это настолько творческий процесс, что никогда не знаешь, что окажется на самом деле полезным. (Михальцова)

Windows versus Linux

В заключение приводится пример того, что все хвалят Windows за лучшую совместимость с периферией, чем у Linux. Однако потом выясняется, что у Linux дела обстоят лучше. А все из-за той же проблемы [***совместимость версий***]. Лично мое мнение, будущее за Linux, ведь темпы развития и свобода распространения заметно выделяют этот вид ОС. Ну и одна из важных особенностей — все везде работает с разными версиями. (Белый)

С потерей части функциональности при обновлении я еще не сталкивалась, слава богу, пока только с удобством интерфейса. Например, выход Windows 8 дал мне сильный толчок для перехода на Linux. К слову, вообще не поняла, где распространено мнение о неподдерживаемости периферийных устройств на Linux. Только возгласы о том, как классно не качать драйвера. (Ровдо)

Честно говоря, с каждым днём все больше и больше хочу попробовать Linux. (Ипатов)

Очевидно, что Windows 7 в 1000000 раз лучше, чем Windows NT. Увеличение требуемой ОЗУ – это техническое требование, так возможности ОС возросли в огромное число раз. (Трубач)

Microsoft Stories

В целом все они [продукты компании Microsoft] ***критикуются*** [автором книги]. Во многом, наверное, можно ***согласиться*** с этой критикой. Например, стремительные обновления NET ставит в трудное положение разработчиков, которым приходиться преодолевать появляющиеся риски и тратить средства на переобучение персонала. Обновленная версия SQL 2005-го года оказалась не доработанной, приложения стали работать медленнее. Windows Vista не оправдала ожиданий и оказалась невостребованной с появлением “семёрки”, так и не успев занять сколь-нибудь значительную долю парка “персоналок” и ноутбуков. Microsoft Office критикуется за смену своего интерфейса, который, как мне кажется, не такой уж плохой, чтобы требовать возможность его изменять. (Борисевич)

Из-за постоянных изменений, так активно производимых Microsoft (они там определиться не могут, или решают зарабатывать на «сыром и вялом»), .NET недозрелая платформа: концепции меняют как перчатки, а это несёт убытков больше, чем дохода (курсы и т.д.). Однако, надо сказать, что тому же Microsoft это, наверное, выгодно, предоставлять возможность переобучения для новой платформы (т.е. всё-таки доход, но для Microsoft, а для компании, которой нужно сотрудников переобучить, это убыток). (Гетьман)

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. (Лебедев)

"Выбор стоит между «изучать новые возможности» и «решать задачи заказчиков». А если изучать, то как не ошибиться с перспективой оказаться у разбитого корыта через пару лет." Наверно, одна из самых актуальных и обсуждаемых тем, так что же всё-таки актуально для изучения. Никогда не угадаешь, что будет перспективным через пару лет. (Михальцова)

Нет вшитой поддержки старых версий и нету возможности выбора интерфейса (если только не проплатить). Да, я помню, лично на себе испытал эти проблемы и скажу, что Word 03 самый стабильный и лучший из MS OFFICE пакетов. (Гетьман)

Изменение интерфейса программ Microsoft Office – отличный показатель того, как люди быстро привыкают к старому. (Михальцова)

Чем автору не угодил Office 2007? В 7-м офисе был значительно расширен функционал. Автор явно любит использовать старые софт и технологии. (Трубач)

[Vista] Потренировались на вас за ваши деньги – в точку! (Гетьман, Щавровский)

Продажа долей Гейтса и Балмера – хм, хм, хм. Что бы это могло значить? (Гетьман)

С этой проблемой [x64 и драйвера] уже разобрались и драйверов хватает. Другое дело, что ставятся они зачастую некорректно и нужно ручками это исправлять. Приятно ли это? Кому как. Мне, допустим, да, приятно, я люблю копаться, я от этого больше понимаю. А вот обычный пользователь по логике вещей пошлёт систему на три буквы.
И посылает. Но продолжает пользоваться ей в силу натренированной беспомощности взять и пересесть на MAC OSX / Linux. (Гетьман)

Хочется пожелать автору удачи с таким отношением к продуктам Microsoft. Если ему настолько все не нравится, то пусть переходит на ОС Linux (Unix, OS X, etc.) и пользуется самым “лучшим” и самым “удобным” софтом.

4. Мнение эксперта

Предложенные темы эссе:

Возможен ли дальнейший прогресс облачных вычислений и чего стоит ожидать в будущем? (Белый А.)

Какие трудности возникают после обновления программного обеспечения? (Борисевич П.)

Что же значит продажа долей Гейтса и Балмера? (Гетьман С.)

Солдаты и генералы софтостроения: армейская и / или рабовладельческая иерархия? (Гетьман С.)

Софтерные фирмы производят услуги? (Гетьман С.)

Облачные вычисления. (Григорьев А.)

Развитие технологий с коммерческой точки зрения. (Григорьев А.)

Новое пришествие CORBA в виде облегчённой её версии (Ипатов А.)

ЭВМ вымерли или вымирают (Ипатов А)

CORBA. Подробнее (Лебедев Н.)

Децентрализация ресурсов. Курс взят. (Лебедев Н.)

Наша служба и опасна и трудна. Как чувствует себя разработчик сегодня? (Лебедев Н.)

Минусы развивающихся технологий (Михальцова А.)

Принципы выбора ОС для ПК (Михальцова А)

Будущее облачных сервисов. (Ровдо Д.)

Почему OS Windows намного лучше OS Linux. (Трубач Г.)

«Ошибки Microsoft» (Щавровский С.)

«Есть ли будущее у SaaS» (Щавровский С.)

Является ли проект продуктом? (Ярошевич Я.)

Оценки конспектов:

Белый А.
Свой простой лаконичный и понятный стиль конспекта. **9/10**

Борисевич П.
Тоже свой стиль ведения конспекта: мнение в начале, затем выжимки из книги (copy-paste). Размер мнения немного смущает (один абзац). **9**

Гетьман С.

Новый подход к написанию конспекта, много комментариев и мнения по авторскому тексту**.** Необычные темы эссе. **10**

Григорьев А.
Похож по стилю на конспект Павла Борисевича, однако мнение находится после выжимок из текста книги. И размер мнения чуть больше. **9/10**

Грушевский А.
Конспект не обнаружен. ??? 0

Ипатов А.
По стилю такой же, как и конспект Антона Григорьева, однако, размер и содержательность мнения оставляют желать лучшего. **8**

Лебедев Н.
Конспект состоит целиком из мнения, выполненного в стиле эссе на прочитанный материал. **10**

Михальцова А.
Свой стиль конспекта (мысль – комментарий - заметка). Однако в этот раз меньше материала, чем обычно. **9/10**

Ровдо Д.
Свой стиль конспекта (выжимка из книги – мнение (pros + contras)). В этот раз как-то скромненько. **9/10**

Трубач Г.
Свой стиль конспекта (pros + contras). Но как всегда сухо. Впрочем, это и есть особенность этого конспекта. **10**

Ярошевич Я.
Свой стиль конспекта. (выжимки очень краткие, ответы на вопросы, мнение). **10**

Отчет 6 (Лебедев Николай)

«Затронутые автором темы не взвали у меня бурных поводов для дискуссий» (Гетьман)

«Этот раздел был труден для прочтения» (Лебедев)

«В последнее время весьма трудно написать какой-то отзыв или найти то, к чему можно было придраться или не согласиться. В этой главе было еще труднее» (Михальцова)

«Так получилось, что эта часть книги оказалась сложной для написания своего мнения» (Ровдо)

1. Общее мнение

Устройство и уровни

Проектирование информационных систем является очень трудным и важным делом. Разделение всей логики на уровни — очень грамотное решение. Во-первых, модульность — хороший подход для сопровождения проекта. Во-вторых, можно нанимать специалистов в конкретной области для достижения наилучшего результата. (Белый)

Есть несколько точек зрения, с которых можно рассмотреть любую автоматизированную информационную систему. В них можно выделить слои, кроме этого даже в самых простых программах различают как минимум два уровня. Если мы наложим слои системы на её уровни, то получим достаточно простую матричную структуру, позволяющую бегло оценить, какой из элементов необходимо реализовать своими силами или же адаптировать уже имеющийся готовый. (Борисевич)

Как можно оспаривать информацию, которая уже устоялась? Стоит только принять к сведению и согласиться с мнением автора, хоть оно и бывает иногда немного резким. (Михальцова)

Я думаю, что все зависит от конкретного проекта, но пример про блоги гуру хорош и, на мой взгляд, справедлив, в бесконечность уходить точно не стоит. (Ровдо)

О выборе архитектуры по статье в интернете

Как перестать лазить по «помойкам»? Воспитывать интерес самообучения в людях, основанный на недоверии к глупостям, незнанию, поспешным и необдуманным решениям. (Гетьман)

Многозвенная архитектура

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. (Лебедев)

Практика управления «кейсами» в IT

Если менеджер толковый и не зря ходил на лекции в студенческие годы, то он воспримет всё это как совет в рамках одной глобальной игры под названием «как утереть товарищу нос», ведь кейсы ничем не отличаются от популярных технологий: всплывают тогда, когда больше не нужны. Ну а если менеджер глуповат, то будет всё делать по «инструкции» и сам проект становится каким-то безжизненным. (Гетьман)

История нескольких #ifdef или перемены 1990-х

Пункт про #ifdef. Зачем это вообще? (Трубач)

Самое важное — пробовать сделать что-то самому, а не смотреть примеры и только по ним работать. Важно понимать, как все устроено изнутри, как работает каждая деталь, каждый механизм. А уже потом на основе имеющегося опыта подбирать оптимальные архитектурные решения. (Белый)

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. (Лебедев)

Интервью, честно говоря, было не всегда понятным из-за упоминания систем, для которых я слишком молода, не сталкивалась, поэтому воспринимать сложно. Однако тут главное сама суть, правильно — совсем небольшая группка толковых людей делает за короткие сроки очень классную штуку. Потому что знают свое дело, а еще потратили 3 месяца только на построение архитектуры и разговоры по теме. (Ровдо)

Я не посчитал необходимым конспектировать эту историю, несмотря на то, что она оказалась интересной. Согласен с мнением автора по поводу архитектуры: я тоже являюсь приверженцем так называемого «тонкого клиента». (Щавровский)

2. Темы для эссе

Управление кейсами в IT (Гетьман)

Необходимые слои корпоративной информационной системы. (Борисевич)

«Современные команды 10-15 человек, вооружённые умопомрачительными средствами рефакторинга и организованными процедурами гибкой разработки, за год не могли родить работоспособный заказной проект, решающий несколько специфичных для предприятия задач.» Причины сложившейся ситуации. (Ровдо)

«Почему MVC столь популярен?» (Щавровский)

“Очень просто делать сложно, но очень сложно сделать просто” (Ярошевич)

“Кто не хочет – ищет причины, кто хочет – средства” (Ярошевич)

3. Мнение эксперта

К сожалению, практически все этот раздел книги, как неинтересный. Некоторые возможно даже разочаровались в книге (об этом говорит содержание конспектов), некоторые его не высказали. Однако увлекательные мысли найти удалось, и темы для эссе очень интересные, хотя их мало. Если автор и дальше продолжит в таком духе, то он, скорее всего, потеряет нашу аудиторию.

Отчет 7 (Щавровский Святослав)

Общее впечатление

В этот раз конспектирующие много раз столкнулись мнениями с автором. Было много поводов поспорить. Очень заметен большой диссонанс. Приступаю к отчету.

Основной лейтмотив

В очередной раз конспектирующих зацепили слова автора о паттернах проектирования. И, что самое интересное, мнение было у всех совершенно одинаковое. Отличались лишь словами:

> «Как говорится в негласном слогане паттернов: «Вашу задачу уже кто-то решал до вас», а это значит, что уже придумано элегантное решение, которое вам необходимо дополнить и свести к вашей проблеме. Тут, как говорится: «предупрежден — значит вооружен». Я считаю, это полезное знание.
>
> Однако, стоит отметить разумность фразы, что лучше паттерны изучать не в самом начале своей карьеры программиста, чтобы на себе ощутить всю глубину идеи, чтобы понять, что ты, возможно, не учел.» (Белый Антон)
>
> «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?» (Лебедев Николай)

Или вот, например:

> «Да, я согласна с тем, что начинающему программисту, который хочет стать действительно профессионалом, лучше сначала самому подумать и понаступать на грабли, а только потом читать упомянутую книгу. Но мне кажется, сегодня разрабатывается столько всевозможных приложений, существует столько всевозможных проектов, что программистов, для которых главное не деньги, а сам процесс и самосовершенствование, немало. Но они же должны как-то решать возникающие проблемы. Книга по паттернам им в помощь, незаменима просто». (Ровдо Дарья)

Согласитесь, очень похожие мнения, не правда ли? Коллективный разум, или сыграл свою роль университет, дающий нам наше образование?
Приблизительно с таким же сходством конспектирующие осветили высказывание Стива Джобса: «Обычные художники заимствуют, великие воруют». Это действительно очень спорные слова, но тем они и интересны. И именно своей спорностью рождают большое количество рассуждений:

> «Очень и очень спорные слова. Не могу прямо сейчас понять, что же мне в этой фразе не нравится: то ли тот факт, что эти грязные слова были произнесены довольно авторитетным человеком, то ли действительная отрезвляющая правота этих слов, сбивающая мои «розовые очки». Трудно сказать». (Гетьман Святослав)

Очень понравились мне еще слова Антона Григорьева:

> «Например, в том же кинематографе никто не любит плагиата. Это сразу бросается в глаза, и это сразу осуждается. Но если позаимствовать элементы из других произведений и при этом уместно и правильно их применить, при этом добавив что-то своё (при чём не обязательно свои элементы, а может всего лишь свой подход, своё виденье, свою любовь и уважение к ремеслу), то вполне может получиться шедевр. (Вспомнить того же Квентина Тарантино)». (Григорьев Антон)

Вообще говоря, высказывания Антона Григорьева в этот раз мне очень понравились. Мы вернемся к ним чуть позже.

И, наверное, третьей по популярности темой стали сборщики мусора, и вообще любого рода проявления «самодеятельности» компьютера. В этот раз мнения разошлись.

Некоторые писали очень достойно, как очень серьезные заскорузлые бородатые программисты:

> «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». (Лебедев Николай)

А были и более либеральные взгляды:

> «А вставка про сборку мусора была, как мне показалось, ни к чему. Да, глава посвящена тому, что надо думать головой, но это не причина для того, чтобы еще раз говорить, что сборщик мусора вещь хорошая, но «настоящие программисты сами следят за памятью». Каждый работает, как ему удобно». (Белый Антон)

По моему мнению очень показательно и правильно получилось. У каждого имеет свое мнение, прекрасно.

Я обещал вернуться к Григорьеву Антону. Антон выразил очень интересные мысли как по поводу SE:

> «Разработка – это творчество. И об этом не надо забывать. Но также это и технический труд. То есть, никто не потребует от вашего фильма отсутствия багов и поддержку Internet Explorer 5. А от проекта потребуют. В этом и сложность, как найти это тонкое равновесие?
>
> Я считаю профессию разработчика очень сложной. Делать выбор в жизни всегда сложно. А когда твоя работа состоит из постоянно выбора «А как сделать лучше всего?», то это совсем тяжко. Но такие люди нужны». (Антон Григорьев)

так и по поводу бытия в целом:

> «Просто человеческое общество так построено. Человек всегда выживал благодаря своим навыкам (и в первобытные времена, и в современны). А сейчас тем более, ведь теперь никто не хочет просто выживать. Все хотят быть признанными, значимыми, необходимыми. Но является ли человек просто набором умений и характеристик? Если да, то всё отлично, ведь само общество требует от тебя их наращивание. А если нет? Не теряем ли мы самих себя в устройстве нашего общества?» (Антон Григорьев)

И как апофеоз:

> «Я считаю, что идеальное общество, это общество, в котором людям не придётся выживать. Только так человек может пойти дальше в своём развитии». (Антон Григорьев)

Неописуемо емко, красиво и правильно, как мне кажется. Спасибо Антону за эти слова.

Темы для эссе

Тем для эссе было не очень много. Более того эти «мало» тем еще и пересекались довольно часто. Наиболее частыми были: «Обычные художники заимствуют, великие воруют», «Thinking in patterns, или приманка для неспециалистов». Еще очень интересными мне показались «Изобретение велосипедов – пустое дело?», «Учитесь у лидеров рынка, их успех не от «просто так»», «Маркетологи в IT сфере» или вот сугубо техническая, каких довольно мало «Типы OutOfMemory в Java и как их избежать». И тема, которая выбилась из общего контекста, но которую я обойти не могу «Потребности человека и их реализация в современном обществе». Автора, я думаю, называть нет смысла.

Заключение

Общее впечатление очень и очень положительное. Писать отчет было довольно легко. Либо потому что это уже третий по счету отчет и рука сама пишет, либо потому что конспектирующие радуют. Очень порадовали высказывания Антона Григорьева, поэтому в этот раз его хочу выделить. И вообще, к третьему отчету я хочу объявить нововведение: начиная с этого, а затем каждый последующий (а это всего лишь один)

отчет я буду выделять несколько конспектирующих. В этот раз этот конспектирующий один, и это Антон Григорьев. Посмотрим, что будет в отчете по десятой части конспекта.

Отчет 8 (Гетьман Святослав)

1. Общее мнение

Сложно не согласиться с автором. (Григорьев)

Автор немного преуменьшает умственные способности начинающего программиста (Ровдо)

Журнал хоз. Операций

Вся глава – завуалированная пропаганда синхронизации. Бесспорно, это очень важная часть (процесс) программирования. После прочтения в книгах о синхронизации, могу сказать, что весьма печально, что применение этого процесса описывается только в экономической сфере. Всё-таки мне было бы интереснее прочитать об использовании в других областях, а также о пагубном влиянии синхронизации, если такое вообще имеется. (Михальцова)

Синхронизация транзакций действительно нужная вещь, т.к. покупатель будет недоволен, если ему продадут не так, как он заказывал, что негативно скажется на репутации магазина. (Трубач)

UML диаграммы и средства для работы с ними

Мне кажется, что это весьма удачное решение. (Белый)

Учитывая то, что "рисование кода" итак достаточно полезно в программировании (я имею в виду изображение задачи на листочке. Ведь так гораздо удобнее увидеть, что требуется, и структурировать дальнейшую работу.), то UML-диаграммы точно получат положительные отзывы. (Михальцова)

UML диаграммы очень часто помогают разобраться что и как должно реализовываться. К тому же сейчас есть много средств их посторения. (Трубач)

[В UML должно настораживать уже самое первое слово – «унифицированный». Не универсальный, то есть пригодный в большинстве случаев, а именно унифицированный, объединённый.] Всё верно. Если абстрагироваться, то можно объединять много разного нехорошего и неполезного. От этого мы результат хороший не получим, сколько бы мы не надеялись на то, что «минус на минус даст плюс» или что какая-то мелочь, затесавшаяся, не вылезет на поверхность. Является ли подобным объединением UML? Нет, во всяком случае, уверен, есть задачи, когда использование UML облегчает производственный процесс или процесс разработки. (Гетьман)

UML. Unified Modeling Language. Языком назвать сложно, а вернее неправильно. Несмотря на слово Language в названии. Да и Unified в названии - обман. UML вовсе не универсальный. Он скорее унифицированный, то есть «объединяющий». Но сути не меняет. Лично я не очень понимаю его важности и не вижу ему применения. Скорее всего это обусловлено моей неопытностью, но вместе с тем. Хотя UML-диаграммы сильно помогают в понимании паттернов проектирования. Что есть, то есть. (Щавровский)

Вообще, я сама всякими утилитами для создания UML диаграмм не пользовалась. Но, конечно, на этапе проектирования не обходилась без листиков с рисуночками, описывающих сущности приложения и их взаимодействие. Мне их хватало, наверное, поэтому я не вижу смысла в проверке системой правильности диаграммы (отсутствие циклов и т. д.) и генерации кода. Но это потому, что я не сталкивалась с крупными проектами. Все-таки в больших масштабах первый пункт может быть очень полезен, сложно уследить за большим количеством компонентов и их связями. А к идее генерации кода я отношусь немного скептически. Мне кажется, нельзя на это полностью полагаться, нужно проверить, что ж там сгенерировалось. Но ведь велик соблазн оставить, как есть, правильно. Так что я бы делала ручками, да, много рутинного кода, но 90% это

повторяющиеся блоки. Зато можно быть уверенной, что в ошибках виноват ты, а не система. Первый вариант менее обидный, как мне кажется. (Ровдо)

Сравнение геоцентрической модели с UML красиво и похоже на правду. (Щавровский)

«Явление зависит от условий наблюдения. Сущность от этих условий не зависит.» Очень интересное заявление, которое, как мне кажется, невозможно подвергнуть критике. Именно этот факт различает гелиоцентрическую модель и геоцентрическую модель. Человечество пришло к гелиоцентрической модели, и теперь отрицание этого факта - абсурд. Но это же самое человечество когда-то пришло к геоцентрической модели, и отрицание было тем же самым абсурдом. Теперь же создан UML. Только в случае UML система не геоцентрическая, а заказчикоцентрическая. А что с альтернативами? Их не много. Наша отрасль сейчас в положении «лучше такой UML, чем никакой.» (Щавровский)

Проектирование

Так же отметилась мысль, что в большинстве случаев при построении моделей мы наблюдаем за явлениями, хотя должны наблюдать за сущностями. Вот и получается, что мы видим все только с одного определенного ракурса. А это может привести к тому, что мы сделаем не универсальную модель или, что еще хуже, вообще неверную. (Белый)

Ошибки в проектировании, паттерны

Меня смущает тот факт, что-то автор ругает паттерны за то, что миру представляются готовые решения, которые дают возможность программисту меньше думать, то сам преподносит на блюдечке это самое готовое решение. (Ровдо)

Антишаблон «таблица остатков». Суть его заключается в том, что приложение использует лишь остатки, не принимая во внимание историю их изменения, что приводит к некоторым сложностям. Надёжнее использовать в приложении журнал операций и вместе с этим хранить вычисленные остатки. Хранить таблицу остатков следует для того, чтобы не ждать окончания их расчётов, которые могут затянуться. (Борисевич)

[Слишком часто формулировка «лучше плохой, чем никакой» стала широко применяться, что настораживает.] На днях была опубликована статья в онлайн-журнале TheJournal, тема которой, если вкратце, то была такая: баги из приложений и операционных систем никуда не исчезнут, придётся с этим смириться, уважаемые граждане. Посему, вспоминается эта статья при чтении формулировки, причём не очень хорошими словами вспоминается. (Гетьман)

Весь прочитанный отрывок можно ужать до определённой мысли: зри в корень. И это похвально. Мне кажется, что автор пытается заставить нас быть разумней, так сказать. Я и сам с таким сталкивался. Когда кажется, что что-то понял (какую-нибудь вещь или систему), но потом оказывается, что недостаточно глубоко и точно. То есть, очень важным качеством я считаю умение видеть суть, вкапываться вглубь, чтобы действительно что-то осознать. Мне кажется, что люди часто просто не осознают степень их непонимания тех вещей, которые, как они думают, они понимают. И так во всех сферах жизни. Не только при проектировании чего-либо. Так что, ещё один плюсик в копилку этой книги. (Григорьев)

2. Мнение эксперта

Темы для эссе

Использование UML в работе. Возможные альтернативы (Белый)

Какие есть известные антипаттерны в разработке учётных программ? (Борисевич)

Лучше плохой, чем никакой? (Гетьман)

Явления и сущности. (Григорьев)

Правда ли мы что-нибудь понимаем? (Григорьев)

UML в наше время. Используется ли по сейчас? (Ипатов)

Таблица остатков. Её основные назначения (Ипатов)

Зачем нужна история операций? (Михальцова)

Отрицательное влияние синхронизации (Михальцова)

Повышение продуктивности «двусторонними» инструментами (two-way tools) (Михальцова)

Two-way tools. (Ровдо)

Синхронизация транзакций (Трубач)

Плюсы и минусы UML (Ярошевич)

Оценки

Белый : 9

Борисевич: 9

Гетьман: 9

Григорьев: 10

Ипатов: 7

Михальцова: 10

Ровдо: 10

Щавровский : 9

Ярошевич : 9

Отчет 9 (Лебедев Николай)

Уважаемый автор книги на мой взгляд слишком увлекается C# и СУБД. (Гетьман)

Опять-таки сложно не согласиться с автором. (Григорьев)

Честно говоря, даже написать нечего. Я даже второй раз пробежала по тексту в поисках чего-то нового. (Ровдо)

К сожалению, для меня темы, описанные на данных страницах, совсем не вызывают интереса. (Трубач)

Интересная жизненная история, недостойная конспектирования. (Щавровский)

1. Общее мнение

Читать отрывки из книги из-за очень тонких и специфически рассмотренных автором вопросов трудно, проблематично понять после всего того, что наворотил автор, к чему же он это всё сказал. (Гетьман)

Я прошу прощения, но действительно в этом отрывке из 23 страниц не смог найти ничего интересного. (Щавровский)

Чтение было достаточно занимательным, однако тяжело конспектировать подобные истории, так как важны мелочи и детали. (Ярошевич)

Когда старая школа молода

Очень поразил тот факт, что данные обрабатывались быстрее (а там их было ого-го сколько), чем читались и гуляли по слоям на пути к конечному пользователю. Все-таки в настоящее время действительно не хватает знающих специалистов, (Белый)

собственное весь прочитанный отрывок можно ужать до определённой мысли: зри в корень. можно ужать до определённой мысли: зри в корень. И это похвально. Мне кажется, что автор пытается заставить нас быть разумней. (Григорьев)

Весьма сложно как-то комментировать прочитанную информацию, в которой либо рассказывается история решения конкретной проблемы, например, упрощение функционала, либо просто обсуждается какая-то технология. (Михальцова)

Архитектура Оптисток выглядит довольно-таки интересно. Так как система является системой автоматизации продаж на выезде, то она является нужным и интересным продуктом. (Трубач)

Раньше программисты знали область, в которой работают

Не могу сказать, что сейчас прямо таки всё-всё-всё не так. Да, то, что раньше было узким, сейчас расширело, набрало массовость, либо исчезло за ненадобностью. Но, тем не менее, необходимость разбираться и делать это самостоятельно никуда не делась. Это одна из первых вещей, на которые смотрят во время испытательных сроков более опытные дяденьки и тётеньки. Да и сами опытные дяденьки и тётеньки на то и гордо носят название «опытные», что знают область не понаслышке, а по работе вживую. А ценнее этого никакая философия быть не может. (Гетьман)

Архитектура сокрытия проблем

очень важным качеством я считаю умение видеть суть, вкапываться вглубь, чтобы действительно что-то осознать. И так во всех сферах жизни. (Григорьев)

Такое явление, на мой взгляд, свойственно не только разработчикам, но и всему современному обществу. (Лебедев)

Как стране не хватает хороших экономистов, так и IT-компании администраторов базы данных (Михальцова)

2. Темы для эссе

Какие проблемы могут возникать, при сильном разрастании базы данных? (Борисевич)

Всегда ли необходимо скрывать проблемы, возникающие во время разработки? (Гетьман)

Явления и сущности. (Григорьев)

Правда ли мы что-нибудь понимаем? (Григорьев)

Как донести технологию до конечного пользователя? (Лебедев)

Развитие и актуальность БД (Михальцова)

Важность документации ПО. (Ярошевич)

3. Мнение эксперта

Не знаю почему, но мне опять достался раздел, который все посчитали неинтересным, а размеры конспектов чрезвычайно малы. Я с завистью наблюдаю за своими коллегами, у который целая куча материала. Причиной тому скорее всего история из жизни автора и его наблюдения, которые на сей раз идут очень глубоко, настолько, что нам не всегда легко это понять. Многие посчитали, что автор стал немного повторяться, даже высказали претензии. Иные отмечали, что читать истории им интересно, но дать свою оценку очень трудно. Все, что есть, можно увидеть выше.

Отчет 10 (Щавровский Святослав)

Общее впечатление

На этот раз (кстати последний) конспектирующие порадовали не меньше, чем обычно. Правда многие столкнулись с проблемой отсутствия опыта и, как следствие, отсутствие мнения по поводу вопросов, поднятых автором. Но это не помешало. Более того, встречались прекрасные моменты. Но обо всем по порядку. Приступаю к отчету.

Основной лейтмотив

Никто, слава богу, никто не стал противиться автору по поводу ревизий кода. Впервые наступило всеобщее согласие:

> «Смысл этой части заключается в том, что каждый уважающий себя и других программист должен поддерживать код проекта в чистоте. Речь даже не столько о клин-коде, хотя это очень важно, сколько о «чистке» старых версий реализаций и функций». (Белый Антон)

> «Ревизия кода. Очень согласна с тем, что начинать ее делать надо чем раньше, тем лучше. Мне кажется, это прозрачно — потом не придется недоумевать, почему не работает функция, и искать ошибку в огромной куче написанного кода». (Ровдо Дарья)

Никто даже не стал спорить с тем, что ревизия кода должна проводится опытным программистом:

> «Ревизия кода нужно не для слежения за выполнением проекта, но и для обучения молодых специалистов. Когда более опытный специалист производит ревизию кода молодого и указывает на ошибки, то это позволяет молодому специалисту поднять свой навык написания кода и предостеречь от появления ошибок». (Трубач Геннадий)

Но, правда, слегка уточнить не постеснялись:

> «По поводу высказывания, что ревизию должен осуществлять опытный программист. Конечно, опытный. Но и проектировать систему должен опытный программист, и реализовывать функционал, и тестировать. Везде нужны опытные программисты, не думаю, что этот факт нужно упоминать отдельно». (Ровдо Дарья)

Конспектировавших очень порадовал тот факт, что методологии, освещаемые автором в десятом отрывке, мы уже изучили и много чего понимаем:

> «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 соответствие новым требованиям 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». (Лебедев Николай)

> «Все перечисленные методологии мы уже рассматривали на занятиях, по водопадной модели даже писали эссе, не считаю нужным много разглагольствовать на эту тему». (Ровдо Дарья)

> «Особенно стоит отметить тот факт, что на темы, поднимаемые в пунктах, мы уже успели написать доклады. Так что все было понятно, и интересно было узнать точку зрения автора на темы, что мы поднимали сами». (Ярошевич Яна)

Вопрос тестирования тоже был затронут, но несколько меньше. Очень явно коррелируется с этим фактом то, что писавшие о тестировании так или иначе знакомы с промышленной разработкой. Ничего странного, все вполне закономерно.

> «Еще слово было сказано по поводу тестов для систем, которые склонны часто меняться. На мой взгляд, тесты должны быть фундаментальны и проверять общую и основную логику приложения, а не какие-то мелочи. Потому как именно на перекраивание тестов под эти мелочи уходит больше времени, чем на написание работающей программы с учетом новых требований и пожеланий». (Белый Антон)

> «Раньше я даже как-то не задумывалась о тестировании как о чем-то конкретном. Да, тесты нужны, конечно, без них никуда — говорили мы все время. А когда начинаешь погружаться глубже, понимаешь, что да, возможно, модульные тесты не так хороши, как хотелось бы. Функциональные тесты сейчас для меня кажутся куда лучше решением». (Ровдо Дарья)

Некоторые писали довольно интересные поучающие истории (на самом деле одну):

> «Давеча беседовал со своим куратором на работе, надо было foreign key-ю атрибутику поменять, чтобы шло удаление. Так вот задал я ему вопрос, как мне всё-таки это дело организовать? В миграции для БД? На что он мне взял и заявил резко, но правильно: нет и ещё раз нет, исключительно в логике программы. Почему? Да потому что начнут ребят использовать другой сервер, или, вот, пользователь захочет заниматься такой чушью, то днями и ночами придётся весь маппинг к БД переписывать и перестраивать. Кому это надо?!» (Гетьман Святослав)

Не знакомые с промышленной разработкой не стеснялись это заявить, что безусловно плюс:

> А в остальном, мне нечего сказать. Ибо весь опыт разработки и тестирования, который у меня есть, вполне является поверхностным. Поэтому спорить с автором (а иногда и понимать) я не могу». (Григорьев Антон)

Правда согласию с автором суждено было когда-нибудь не состояться, чему нельзя не радоваться. Притом разрушилось согласие очень эффектно:

> «Намотав на ус всё вышесказанное, взгляните ещё раз бегло на авторский текст. Что это за «вполне программистский коллектив»? Ребятки пишут софт на коленке для шарашкиных конторок? Если только так, ибо компании покрупнее, у кого в штате есть хотя бы один человек, который сечёт в чём-то кроме программирования, смекает, что надо бы как-то эти залежи поудобнее разгребать. Да и стартапщики сейчас не такие уж и забулдыги, чтобы не пользоваться СКВ. Напротив, это модно и приятно пользоваться системами контроля версий». (Гетьман Святослав)

На этом обсуждения книги закончились. Но интересные мнения не закончились. Мне показалось интересным следующее мнение:

> «Понятия не имею почему, но сейчас, я представил себе it-сферу этакой мамашкой, у которой миллионы раскрытых ртов, которых надо кормить, причём кормёжка - это отбор еды у других детёнышей. Так ладно, тут есть эти, вокруг ещё больше рождается, а мамашка не справляется и начинает тихонечко сходить с ума. А много детёнышей появляется, потому что о мамане много всего такого хорошего говорят, все вокруг её расхваливают. Ну и от мужиков-отраслей отпора нет. И получается, наверное, что от этого мамашка, которая была более-менее сконцентрирована в себе, в университетах и философско-математических концепциях, решила пошалить. Отчего и стала вести себя, как проститутка – прыгать на каждого и рекламировать себя не лучшей стороны, забыв о том, какой была изначально». (Гетьман Святослав)

Прекрасное сравнение.

Темы для эссе

Тем для эссе было еще меньше, чем было в прошлый раз. Но они почти не пересекались. Вот самые интересные из них: «Как выбрать лучшую методологию для конкретного проекта?», «Тестирование - пациент жив или мертв?», «Как помочь it-сфере двигаться в полезном направлении?»

Заключение

Мнений в этот раз было не очень много, но они все были очень интересны. Это последний отчет. В прошлый раз я обещал выделять несколько человек. В этот раз это Святослав Гетьман. На этом все. Последний отчет закончен.

Отчет 11 (Гетьман Святослав)

5. Общее мнение

На протяжении заданного куска текста автор рассказывал о приложении, состоящем из набора слоёв трёхзвенной архитектуры на основе веб-служб. То есть прочитанный материал воспринимается просто как история, констатация факта, а не как материал-почва для дискуссий и рассуждений. В общем-то идея обсуждения книги и сама книга достаточно неплохи, вот только автор зачастую уходит куда-то вглубь материла настолько, что уловить суть происходящего способен только человек, непосредственно имеющий или имевший дело с рассказанным. (Михальцова)

В данной части главы было представлено много практического руководства по использованию Genie Lamp, поэтому я это пропустила, оставив саму суть. Не знаю, честно говоря, почему это так подробно было расписано, мне кажется, это не та книга, которую будешь читать, чтобы научиться работать с Genie Lamp. Для того, чтобы показать, что это все делается несложно, мне кажется, пункт можно было сократить. Но ладно, это авторское право :) (Ровдо)

Корпоративные решения от Microsoft

И снова автор рассказывает о том, как лень и недальновидность разработчиков (а также неповоротливость крупных компаний) приводит к неоправданно большому количеству лишней работы. И снова мы обещаем себе быть хорошими разработчиками. (Григорьев)

Какие же всё-таки Microsoft потешные. И не они одни такие, просто такого рода «фейлы» (fails) прочно ассоциируются именно с ними, с их продукцией. И, тем не менее, их продукция остаётся каким-то призрачным гарантом качества, благо, дизайн уже медленно подыхает.

А вообще, для меня мораль сей басни была такой: корпоративное решение другому корп. решению рознь. А мелкософты продолжают мутить воду, от которой страдают и разрботчики, и весь проект в целом (от простоя). (Гетьман)

По сути расписан один из возможных вариантов работы. Кратко его можно описать так: эту технологию можно, а ту нельзя; так, почему-то эта штука не настраивается, поищу-ка я в Google; О, вроде нашел, но оно все равно не работает:(здесь такое понятие, как «танцы с бубном», т.е. попытки хоть как-то заставить работать нужную вещь; ну и завершается облегченным – «ну наконец-то!» Вот такая поучительная история:) (Трубач)

Автоматизация

Автоматизация написания кода по прототипу (модели) проекта - это весьма удобный инструмент, особенно если объемы столь велики, что эта рутина заставляет тратить слишком много полезного времени. Но опять же и он не идеален, а именно возникают проблемы, когда мы начинаем смешивать модель и ручное написание кода поверх сгенерированного. Хотя я слышал, что современные системы более проработанные и функционал существенно улучшился. Лишний раз подтверждается мысль, что лень — двигатель прогресса. (Белый)

[Лень — двигатель прогресса] Ой, прямо в точку! Ради этого, собственно и нужна автоматизация. Кому нравится создавать кубики с полного нуля для того, чтобы построить небоскрёб? Чёрт возьми, да кто-нибудь, занимаясь веб-разработкой, начинает с реализации сетевых протоколов и описания структур в плюсах через код ассемблера? Знать, как устроены кубики необходимо, но очень вредно собирать то, что уже давно есть и широко используется, с абсолютного нуля на реальном производстве. ~~Жопа горит.~~ Сроки горят. Да, это необходимый этап, если ты создаёшь что-то инновационное или же учишься (вспомнить хотя бы УП). Но иначе это просто трата времени. Надо

просто уметь разбираться в документации так, чтобы БЫСТРО найти необходимое и это применить. (Гетьман)

Автоматизация - хорошая штука, но не стоит ей увлекаться. Да, здорово иметь под рукой инструменты, которые позволяют быстро соорудить каркас обычного приложения (без всяких наворотов, чисто модели, контроллер, вьюхи – в зависимости от используемого паттерна своё наполнение) и работать с порождённым шаблоном, подстраивая его под себя. НО надо не ограничиваться одним шаблоном, смотреть шире и знать, то кроме MVC есть хотя бы MVVM и, соответственно, знать, как ручками, в случае высадки на местность, кишащую аборигенами, написать приложение с выбранным паттерном, чтобы спастись. (Гетьман)

Наверное, это действительно помогает, когда у вас серьезная команда, занимаетесь серьезным большим проектом. Но вот лично мне, как начинающему разработчику, нравится больше вот такой вариант: просто возьмите и спихните эту грязную работу на джуниоров. Серьезно, вообще не смущает то, что это звучит как эксплуатация. Мне кажется, любой программист все-таки должен уметь написать то, что сгенерируется по модели. Для джуниоров это отличная возможность разобраться. (Ровдо)

Интересный вопрос. Автор совершенно правильно (ещё бы) выделяет проблемы и недостатки современных систем такого рода. Но что будет, если эти недостатки исчезнут. Если эти системы станут совершенными и будут генерировать идеальный код (хотя не факт, что такое вообще возможно). Ясно, что в таком случае кодеры исчезнут, как таковые. Но насколько изменится профессия разработчика? Полагаю, что она станет более доступной, снизится порог вхождения.

И ведь такие тенденции, по всей видимости, наблюдаются с самого появления профессии разработчика и IT-сферы вообще. С развитием технологии становятся всё проще для работы и применения. Так и сами программисты из университетской элиты постепенно стали вполне рядовыми рабочими. И что, если в будущем их самих заменят программы, которые будут писать программы? Так и в сфере промышленности люди прошли путь от ремесленников до промышленных роботов.

С древних времён люди применяют известные им знания и технологии для упрощения своего труда, вплоть до того момента, пока эти самые технологии полностью не заменяют человека. Ждёт ли то же самое разработчиков и программистов? Верно ли говорить о деградации профессии? Или всё-таки об её развитии? Мне, на самом деле, хочется быть оптимистом в этом вопросе. Сам я считаюсь будущим разработчиком, и не хочется думать, что я, не начав свой путь, уже стану ненужным. К тому же я уверен, что многие скажут мне, что человека не так уж и легко заменить в настолько интеллектуальных занятиях, как разработка чего-либо (но ведь ИИ не дремлет).

Но, в общем, я считаю, что это правильный путь. По моему мнению, человечество должно достичь такой стадии, когда человек сможет всю свою жизнь заниматься, скажем так, возвышенными и полезными вещами. А именно: творчеством, исследованиями, преподаванием, развитием и т.д. и т.п. То есть человек не должен стоять за станком, производя продукты потребления. Человек, будучи существо, способным осознавать себя и окружающий мир, должен заниматься чем-то более важным.

Но, возможно разработка как раз-таки является чем-то важным. Всё-таки она вполне связана с исследованиями. Да и является творческим процессом, как мы выясняли раньше. В общем, я не берусь однозначно отвечать на этот вопрос.

При этом, в тексте есть оговорки про то, что средства автоматизации призваны не к тому, чтобы лишить разработчика работы, а к тому, чтобы избавить его от рутины, оставив больше времени для действительно профессионального и творческого труда. Как говорится, рутина убивает. Если

смотреть на подобные системы в таком свете, то, пожалуй, все разработчики должны обеими руками быть за них. (Григорьев)

Генерация кода

Генерация кода по моделям - идея схожая с автоматизацией написания кода по модели, однако мы не набрасываем элементы на панель, а пишем конфигурацию модели в XML. Как утверждает автор, мы можем 600-ми строчками XML кода заменить около 30.000 строк кода на SQL, Java, XML для разных слоев нашей архитектуры. А также написанный таким образом код, вернее сгенерированный, не нуждается в тестировании. Как итог, наша производительность возрастает многократно. (Белый)

Идея про модель «Лампа, полная джиннов» интересна тем, что она проста. И к тому же часто возникают мысли про что-то, что будет само генерировать код. Эта модель так же завлекает тем, что, как утверждает автор, соотношение числа строк мета-кода описания модели к коду его реализации на конкретных архитектурах и платформах составляет около 1 к 50, что не может не радовать, так как писать нужно намного меньше кода. (Трубач)

Лампа, полная джиннов определенно меня заинтересовала, однако, на первый взгляд, мне все же не видится эта технология технологией, которой имеет смысл пользоваться всем. Но для генерации рутинного, одинакового кода - идеальное решение.

Но идеи разрабатывать программы, минимизируя стадию кодирования я не очень понимаю. Мне кажется, что таким образом добиться качественного софта не представляется возможным. Палки себе же в колеса. Однако без таких идей, как мне кажется, прогресс шел бы несколько медленнее. В частности, в управляемой моделями разработке или в программной фабрике существует возможность сгенерировать код, который сразу будет работать, а дальнейшее проектирование можно продолжать взаимодействовать с готовой моделью, что действительно интересно. Мне очень понравилась аналогия с «мужиком с лопатой» и «экскаватором». (Щавровский)

Мне была интересна программная фабрика, в которой рассказывается про инструменты CASE, позволяющие генерировать код, скомпилировав который, можно сразу получить работающее приложение. Первые CASE инструменты, выросшие из редакторов графических примитивов, были представлены в 1980-х годах. Можно заметить некоторые недостатки CASE-средств:

1) Если ручное написание кода принять за максимальную гибкость, то CASE может навязывать стиль кодирования и шаблоны генерации частей программ.
2) CASE работает только при условии, что ручные изменения генерируемого кода исключены или автоматизированы.

В качестве решения перечисленных проблем появились так называемые двусторонние CASE-инструменты, позволяющие редактировать как модель, непосредственно видя изменения в коде, так и, наоборот, менять код с полу или полностью автоматической синхронизацией модели. (Борисевич)

А вот это интересный и полезный подход. Во многом, он зависит от платформы, на которой выстраивается взаимодействие с базой, моделью и так далее. Если платформа очень перенасыщенная деталями, то от проектирования она будет отвлекать на хотфиксы и консультации с ребятками из microsoft. Иначе можно спроектировать не совсем гибкий по функционалу слепок будущей системы, которую можно уже переносить со всеми оговорками на более изощрённую платформу. К примеру, начать разработку на RubyOnRails и перенести всё на Java. (Гетьман)

Автоматизировать процесс разработки круто, как говорят, «базару нет», но, не стоит забывать, что смысл работы программиста не в том, чтобы обойтись нажатием 2-3 кнопок и получить готовый результат. В первую очередь, программисты, это люди, которые шевелят мозгами, ищут и исправляют ошибки, радуются написанию хорошего кода и постоянно совершенствуют свой стиль. О каком тогда совершенствовании может идти речь в таком гиперболизированном случае, когда автоматизирован весь процесс разработки? Единственное, это создавать автоматы. Сначала для программ / продуктов, а затем и для создания самих автоматов. Вот вам и рекурсия, круг замкнулся. Хорошо это или плохо? С моей точки зрения, скорее хорошо, ибо придётся напрягать мозги ради изобретения автоматов и совершенствования изобретённых, но всё равно, область применения ума может заметно сузиться, а это огроменный минус. Надеюсь, такого никогда не произойдёт, хотя бы потому, что автоматизировать всё нельзя. (Гетьман)

6. Мнение эксперта

Темы для эссе

Лень — двигатель прогресса. Примеры из IT.» (Белый)

Эффективность использования средств генерации кода в разработке программного обеспечения. (Борисевич)

Автоматизация глазами живого человека (Гетьман)

«Гибкость» крупных компаний. (Григорьев)

Пределы развития человечества. (Григорьев)

технология программной фабрики (Ипатов)

пакет SVN. TFS (Ипатов)

Книжки по программированию для обсуждения (Михальцова)

«Junior developer и грязная работа» (Ровдо)

Системы, которые генерируют код из какого-либо мета-кода. (Трубач)

Различия кодера и программиста (Щавровский)

Тестирование тестов. Важность тестирования. Сколько стоит уделять этому времени? (Ярошевич)

Важность средств автоматизации (Ярошевич)

Оценки

Белый: 9

Борисевич: 8

Гетьман: 10

Григорьев: 10

Грушевский: 7

Ипатов: 7

Роадо: 9

Трубач: 9

Щавровский: 9

Ярошевич: 9

Отчет 12 (Лебедев Николай)

Эта часть, как ни странно, была самой интересной из всех в этой книге. (Ровдо)

1. Общее мнение

Можно сказать, что сам процесс программирования – это постоянное проектирование (но, конечно же, не такого уровня, как проектирование всего проекта). В этом, к слову, и большой плюс профессии программиста: профессия одна, но каждый раз всё новое. (Григорьев)

Говорят, что ошибаются все, но сложно представить, что на это способно "такое богатое и «багатое» учреждение, как Microsoft. Хотя было бы забавно взглянуть на код и при обнаружении ошибки с довольным лицом сказать: "Господин Гейтс, а ваши программисты накосячили тут". (Михальцова)

Софтостроение и писательство

Очень занятным является сравнение софтостроения и писательства. Собственно, с тем, что они схожи, я согласен. Ведь что такое программирование? По сути, это составление чего-то большого и сложного из множества чего-то маленького и простого. То есть, взять какие-нибудь простейшие объекты, которые выполняют простейшие действия, понять, как они могут взаимодействовать, и выстроить из них систему, способную делать что-то сложное. В этом и проявляется красота, или искусность. И ведь в литературе то же самое. И при всём этом, цель у обоих процессов одна: составить общую, целостную, рабочую картину. (Григорьев)

О «must have»

А такое вообще везде, развелось много таких крикунов-максималистов. (Гетьман)

Считаю, что каждый должен заниматься самообразованием. В каждого это надо вложить. Иначе, человечество просто погибнет от жажды наживы и развлечений. (Гетьман)

Об организации работы

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. (Лебедев)

Очень легко попасть в «конвейер» и работать в нем лет 30, в котором о софтостроении полностью нет ни у кого представления. (Лебедев)

Все всё «гуглят». (Трубач)

Критика Microsoft мне очень не понравилась. Я помню, что автор верстал книгу в LibreOffice — бесплатный офис. Но верстка — ужасна. Хотя может в печатной версии все довольно-таки неплохо. (Трубач)

Зачем вкладываться в науку, быть меценатом, если это принесёт столько же, а может и меньше, денег, чем если о науке даже не думать. В результате на первый план выходят люди, способные за короткое время и сравнительно небольшие деньги удовлетворить спрос на рынке. Такая тенденция сказалась и на технической литературе. Появление термина "гуглизация" яркое тому доказательство. (Щавровский)

О книге

На самом деле, после прочтения осталось желание вернуться к этой книге позже (возможно намного позже), потому что есть ощущение, что я понял далеко не всё (наверняка понял далеко не всё), что хотел сказать автор. Полагаю, нам ещё предстоит узнать на собственном опыте о том, в каком состоянии сейчас находится IT-сфера. А также точнее понять, какой путь она прошла. То есть надеюсь, что когда-нибудь нам удастся составить в голове целостную картину всего этого. Что касается меня, пока что это точно не так. Но фундамент однозначно положен. (Григорьев)

Довольно-таки познавательная книга, хотя и со специфическими моментами. Если же оценивать в целом данную книгу, то информация довольно таки полезная и в некотором смысле даже жизненно необходимая для начинающего разработчика. Было описание маленькой фирмы, у которой были такие же проблемы, что и у «больших». Что мне лично нравиться в этом деле так это неограниченная возможность, своего рода свобода. Практически любая маленькая компания может “стрельнуть” главное- идея и инвестор. Закончить хочется фразой одной из глав этой книги: “Зато вы можете быть спокойными: без работы в ближайшие десятилетия не останетесь…”. (Ипатов)

Хотелось бы подвести итог из всего этого. После прочтения у меня осталось ощущения, что я лично беседовал с 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. (Лебедев)

В целом, мне эта книга не очень понравилась. Автор показался наглым и заносчивым :) Но тем не менее, там есть пункты, которые стоит будет перечитать в будущем. Я вообще думаю, что конкретно мне просто немного рано ее читать. В книге много моментов, над которыми надо подумать, но, когда ты с этим не сталкивался, все это не так веско. Может быть, через 5 лет я вспомню об этой книге, будет интересно :) (Ровдо)

Автор рассказывал обо многих насущных вещах в софтостроении и реальных попытках их исправить. Где-то удачно, где-то через «костыль», где-то неудачно, но главное, что попытки были. Так, конечно, книга была полезной. Но была бы еще полезнее, если бы в ней не было большого количества воды. (Трубач)

Прочитав книгу до конца, я слегка изменил своё мнение об авторе. Он мне больше не кажется очень недовольным жизнью стариком лет 80, повидавшим очень многое и имевшим в виду чужое мнение. После почтения любой книги я чувствую легкую грусть. И эта книга, как ни странно, не стала исключением. Она оставила некий след в моих мыслях. И это совсем не так книга, о которой предостерегал автор. Но придя домой я её точно удалю. (Щавровский)

2. Темы для эссе

Какой должна быть система образования сейчас? (Гетьман)

Что выбрать: трёхмесячные курсы или четырёхлетнее образование? (Григорьев)

Картина мира в голове человека. (Григорьев)

Влияние “гуглизации” на современное общество. (Ипатов)

Зависимость опыта от возраста (Лебедев)

Фрагментарное мышление. (Ровдо)

«Гуглизация» как помеха получения систематизированных знаний (Ярошевич)

Трата здоровья программистов во время дедлайна, как с этим бороться? (Ярошевич)

3. Мнение эксперта

Вот и отчет от экспертов написан. К сожалению, к нам на почту не дошли все письма, поскольку каждый, наверное, немного расслабился. Все те, кто прислал, проделали громадную финальную работу. Каждый из конспектов был очень содержательным и включал короткое мнение о книге, их можно найти чуть выше. Действительно, очень много интересных мыслей было сказано и финал книги вышел замечательным. От себя могу добавить, что сама работа над книгой была очень интересной и познавательной, а писать отчеты, анализировать и обсуждать внутри экспертной группы было даже весело.