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 молодо- му 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 посте- пенного 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, 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.

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. Бьёрн 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 по- чти на 40 файлов. Вот тебе и вызвал функцию…». 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.

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 це- почка «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». Несогласен со многим. С настолько многим, что перечислять не стану. Интересной темой для эссе мне видится «Почему MVC столь популярен?».

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 архитектура 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 с 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. Тем для эссе в данном отрывке я также не вижу.

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 снежная 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.