Конспект книги

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).

**Темы эссе**

Было бы интересно написать эссе на какие-нибудь отстранённые темы. Например, является ли софтостроение, программирование и т.д. искусством. Или суть и различия мотивации и стимуляции.

**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.

**Темы эссе**

А по поводу эссе, было бы очень интересно сделать эссе на тему парадигм программирования и подхода к написанию программ. Потому что замечаю за собой, что не совсем понимаю разницы и применимости различных парадигм.

**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.

**Темы эссе**

Было бы интересно написать эссе на тему ООП. Разобраться в этой парадигме, её достатках и недостатках. И понять, почему она стала такой популярной.

Или же на тему создания языка программирования, сконцентрировав внимание на том, что должно лежать в основе языка (математическая модель и так далее).

Сравнительное эссе про SQL и noSQL базы данных, отличия и преимущества.

**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 использования. 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.

**Темы эссе**

Облачные вычисления.

Развитие технологий с коммерческой точки зрения.

**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.

**Темы эссе**

Насыщенные интернет-приложения.

Потребности человека и их реализация в современном обществе.

**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.

**Темы эссе**

Целесообразность использования шаблонов проектирования.

Эволюция профессии разработчика (опасность её упрощения).

**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.

**Темы эссе**

Явления и сущности.

Правда ли мы что-нибудь понимаем?

**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.

**Темы эссе**

Зависимость программных технологий от аппаратных.

Что значит быть специалистом?

**Тема 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.

**Тема 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.

**Темы эссе**

«Гибкость» крупных компаний.

Пределы развития человечества.

**Тема 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.

**Темы эссе**

Что выбрать: трёхмесячные курсы или четырёхлетнее образование?

Картина мира в голове человека.