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 «политтехнический 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.