Набор серебряных пуль

Справочник удачных
проектных решений
при разработке ПО

Берлинский Константин

Посвящается моей семье,
в которой я всегда чувствовал
опору и поддержку

Версия текста: 1.37
20/06/2004

СОДЕРЖАНИЕ

1. Аннотация

Войны ИТ-методологов не затихают. Каждые несколько лет нам преподносится совершенно новая, быстрая, легкая, простая, эффективная методика (или новая версия «старой»). И уж она наконец-то решит главную проблему – построение качественного ПО в срок.

Я думаю, что правда о методологиях заключается в том, что их не существует…

Есть лишь УПР – удачные проектные решения – которые могут сработать (или нет) в конкретной ситуации и проекте. Цель этого справочника – собрать их вместе, дать им краткое описание, и подвигнуть ИТ–сообщество к дальнейшему их поиску и классификации…

2. Введение

«Ну вот!» – скажете Вы, прочтя заголовок данной книги. «Ещё один новоявленный пророк - самозванец учит всех жизни, как нужно выполнять программные проекты! У нас и так есть методология, которая отлично справляется со всеми проблемами. Мы адаптировали её под свои нужды, и вроде бы проблем стало меньше…»

И Вы будете правы, … но наполовину. Я ни в сколькой мере не считаю себя новоявленной мессией, который «наконец-то расскажет, как добиться успеха». Но меня действительно интересуют методы эффективной разработки ПО (и как следствие этого знания – повышение своего профессионального мастерства разработчика ИС).

Эта книга не является обоснованием в письменном виде против какой-либо определённой методологии. Хотя ранее, на форумах тематических сайтов, я позволял себе резкие высказывания по поводу различных новомодных методик разработки, которых с религиозным пылом фанатиков отстаивали их ярые приверженцы.

Вы не найдёте здесь и доводов в пользу какой-нибудь определённой методологии, как можно было бы предположить исходя из того, что я долгое время являлся ревностным сторонником методологии RUP, активно занимался её изучением и внедрением в деятельность компании, в которой работал. Причем, в тот момент времени, во мне действительно была уверенность в том, что жесткое разделение участников разработки по ролям (и соответствующим функциональным обязанностям) сможет сделать процесс разработки управляемым и

успешным, а его участников – более удовлетворёнными своей работой.

В связи с этим, своей книгой я рискую навлечь на себя праведный гнев сторонников и защитников всех ныне существующих (и которые появятся годами позже) методик разработки ПО. Однако каждый должен иметь возможность высказать своё мнение, и я воспользуюсь этим правом.

Я уверен, что правда о различных методологиях заключается в том, что их не существует...

А теперь, приведите в чувство всех упавших в обморок, и признайтесь самому себе – что представляет собой сверхновая методология разработки ПО, о которой Вы узнали из последнего маркетингового заявления неважно какой корпорации? Или та методика, которую Вы уже используете в своей повседневной работе, и в которую вложено огромное количество ресурсов (учебные материалы, курсы для ведущих специалистов с выездом в другой город/страну, и, наконец, самое ценное – время)?

Вы думаете, что методика служит организующим фактором разработки, что она четко и ясно говорит, как нужно работать, чтобы добиться успеха в ИТ-области. И, наконец, Вы надеетесь получить конкурентное преимущество, «пуская пыль в глаза» потенциальным заказчикам малопонятными для них фразами типа «мы находимся на 6-ом уровне CMM», «реинженеринг бизнес-процессов», «автоматизация хаоса путем выделения ролей в альтернативных деятельностях» и т.п.

Однако, внедрение одной и той же методики в разных организациях и проектах (а зачастую и в фазах разработки одного проекта!) даёт почему-то совершенно различные результаты. То, что в одних случаях работало хорошо и является стимулирующим фактором, в других – наоборот, тормозит разработку.

В чем же секрет, спросите Вы? Я думаю, что успех проекта зависит от двух факторов:

1) доступные ресурсы (в первую очередь, это качество разработчиков, а второе – это время);

2) способ их взаимодействия.

Если есть доступные ресурсы, и они взаимодействуют максимально эффективным способом, то я считаю, что проект имеет значительно больше шансов родить что-то действительно стоящее.

Что касается методологий – то мне кажется, что все они описывают конечный набор различных способов эффективного использования ограниченных ресурсов. Моя точка зрения состоит в том, что число этих способов (удачных проектных решений - УПР) бесконечно и не нужно ограничивать себя только подмножеством их, в рамках методологии X. Если мы хотим продвинуться в плане успешной разработки программ, то нужно собирать эти решения (аналогично паттернам проектирования) и учиться применять их в нужный момент. Эта книга является попыткой собрать известные мне методы успешной разработки в одном месте.

Приятного чтения и да прибудет с Вами великая сила!

3. Зачем эта книга была написана?

Эта книга была написана для того, чтобы собрать в единую коллекцию то, что я называю «золотые крупицы знания», распылённые по многочисленным источникам, таким как Интернет, литература и просто народное творчество. Фраза «одна голова хорошо, а две лучше» и принцип «разделяй и властвуй», известный еще со времен Римской империи, сделали для развития программной инженерии больше, чем Microsoft и IBM вместе взятые.

После прочтения любой книги, статьи или сообщения на форуме, меня всегда интересовало – что конкретно этот источник может дать мне полезного? Содержится ли в нём какая-нибудь новая мысль, неизвестная мне ранее? К счастью сказать, в большинстве случаев новые идеи присутствовали, но, к сожалению, были основательно «разведены» посторонней информацией, только замутняющей суть дела.

Поэтому, я старался, по мере возможности, после прочтения очередного труда, составлять его «конспект» с перечнем того, какие основные идеи (неизвестные или мало-очевидные в тот момент для меня) пытался высказать автор.

Например, вот такой результат «препарирования» у меня получился от книги [3]:

1. описание бизнес-процесса в виде текста по объему намного меньше, чем в виде графики (это действительно так – самой большой проблемой при работе с диаграммами было то, что рабочий

принтер не поддерживал печать на формате А3 и А2…)

2. заказчики быстрее прочитают текст, поймут и подпишут его (согласятся с ним или выскажут свои претензии), чем выучат UML (у меня, например, много времени уходило на объяснение стрелки include/extended для связей между вариантами использования)
3. нового сотрудника можно быстрее научить писать текст в формате прецедентов Коберна, чем заставить *правильно* использовать UML и продукт, его поддерживающий (например, Rational Rose – графический дизайнер которой оставляет желать лучшего)
4. хорошая классификация целей – всегда нужно работать на одном уровне целей. Уровень повышается, если задать вопрос «Почему?» и понижается, если задать вопрос «Как?» (это большое искусство – быть на нужном уровне цели - диаграммы получаются мелкие и общие, или слишком большие и детализированные)
5. отличный, интуитивно понятный шаблон описания прецедента (основной сценарий из 10 шагов без «если» + расширения основного сценария + дополнительная информация)
6. хорошая идея об использовании средств многомерного анализа собранных требований (например, с помощью электронных таблиц) –

применение сортировки, группировки по различным атрибутам (важность, срок, модуль, роль)

7. и, наконец, на мой взгляд, самое важное – «чем ниже уровень описания целей, тем менее полезными и наглядными становятся диаграммы». Из этого следует идея о «разделении труда» между различными CASE-средствами: для составления диаграмм высокого уровня (сценарий бизнес-процесса, схемы движения информации, диаграмма состояний основного документа системы, взаимодействие программных модулей) использовать инструменты, эффективно применяемые для рисования диаграмм, их связи друг с другом (Rational Rose, MS Visio), а для более детального описания применять «прецеденты по Коберну».

Должен признаться, что написание этой книги было несколько авантюрным занятием – всегда можно будет получить «укол в спину» от какого-нибудь «мастодонта» с 20-летним стажем разработки и отдавшего большую часть своей жизни продвижению методологии X, в виде фразы «сделай столько проектов, сколько я, а потом будешь предлагать решения под своим соусом…».

Мне же, с 24-мя годами общего возраста (из которых всего 3 - опыт коммерческой разработки ПО) и 5-ю реализованными проектами, «по общепринятым правилам» нужно подождать лет

до 40-ка, а потом уже излагать свои мысли в письменной форме.

Но к черту все эти правила! Нужно жить сегодняшним днём, и если тебе представилась возможность «прыгнуть выше головы», то нужно делать это *здесь и сейчас*. Вероятность поднять здоровенный железный шкаф ничтожна мала (см. фильм «Пролетая над гнездом кукушки»). Но это вообще никогда не удастся, если не сделать хотя бы одну попытку.

Мне *действительно* интересно то ремесло, которой я обладаю, и каждый день я стараюсь сделать свою жизнь (и профессиональную деятельность) как можно лучше и счастливее.

Конечно, на самом деле это несерьёзно – думать, что написание книги, статьи или прочего «пиара самого себя» увеличит мою или чью-то ещё образованность (подробнее, см. статью [2]). Однако всё же это имеет смысл.

Написание книги увеличивает мою «виртуальную образованность» (т.е. то, насколько хорошо меня *воспринимают* потенциальные и текущие работодатели). Всё это (встреча с новыми людьми, познавание новой информации) увеличивает мои шансы повысить свою «реальную образованность» (т.е. та *реальная польза*, которую я приношу проектам, в которых участвую). Повышение «реальной образованности» толкает меня на публикацию нового материала. И так далее.

Этой книгой мне хотелось доказать (в первую очередь самому себе), что мир проще, чем кажется на первый взгляд и добиться в нём успеха действительно реально, независимо от тех

препятствий, которые постоянно возникают на нашем пути.

Есть такая байка про миллионера, который сказал: «Я могу рассказать вам, каким образом я заработал каждый свой миллион. Кроме первого». Действительно, если воспринимать то, что называется «жизненным успехом» в качестве восхождения наверх по бесконечной лестнице, то наиболее трудно сделать именно первый шаг. Я постарался сделать для себя эту книгу «экзаменом» на право подняться на первую ступень.

Ну и, наконец, это на самом деле огромный кайф – писать книгу. Аналогично сочинению музыки, рисованию картины или вырезанию скульптуры. Красота на самом деле «спасёт мир, меня и программные проекты». Когда ты видишь, как обрывки мыслей и фраз, после терпеливой обработки (мучительного поиска синонимов и выделения деепричастных оборотов) превращаются в связный текст с логичным сюжетом, испытываешь ни с чем не сравнимое удовольствие.

Программирование тоже дает такое ощущение. Это связано с тем моментом, когда разрозненная информация, требования, приказы, инструкции, слухи и досужие домыслы, относящиеся к системе, как огромные неповоротливые булыжники вдруг складываются у тебя в голове в единую скульптуру. Каждый пазл становится на своё место и остаётся только брызнуть на них живой водой. Каменная статуя просыпается и рукотворное существо начинает делать свои первые шаги…. Это одна из самых больших радостей нашей профессии.

4. Что было задумано

Изначально, у меня возник следующий замысел относительно книги (планировалась только «бумажная» версия). В начале, сделать обзор современных методологий разработки ПО. Затем высказать основную идею о том, что все методологии используют подмножество из общего пространства УПР (см. рис. 1).

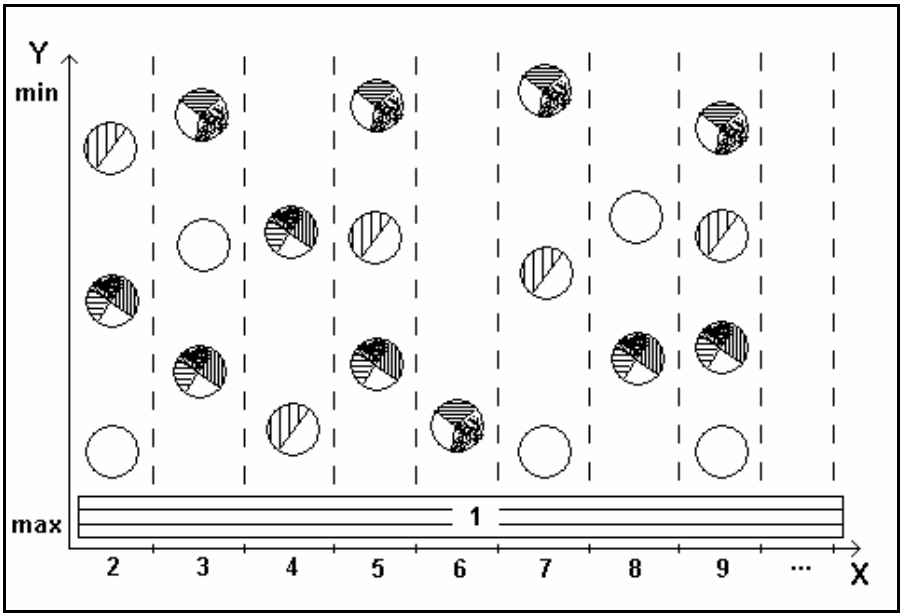

Рис. 1 – Единое пространство УПР.

X – ось этапов ЖЦ ПО;
Y – степень важности УПР для конкретного этапа;
1..9 - ЖЦ (1-фундамент проекта - «Управление»);
Круги – УПР, известные на данный момент;
Виды штриховок – различные методологии разработки ПО, включившие решение в перечень своих постулатов.

В середине – дать пятьдесят чистых листов. Каждый лист представлял бы собой форму для ввода удачного решения, найденного читателем (скорее писателем) в процессе разработки очередного проекта. Форма имела бы стандартную форму: название, код этапа ЖЦ, оценка эффективности, описание и дополнительные источники информации по решению.

Заканчивалась бы книга главами «Содержание» и «Библиография». Естественно, они бы тоже заполнялись вручную. У каждого есть свой «золотой набор», состоящий из книг, WEB-ресурсов и телефонов ближайших пиццерий.

Идея была в том, чтобы каждый написал свою собственную книгу и даже вписал свои инициалы на обложку.

Но затем я просмотрел свою домашнюю библиотеку, и обнаружил, что во многих из них идея о «дополнении» книги читателем уже реализована. Это так называемые листы «для заметок». Заметки у меня отсутствовали…

Поэтому, я решил подойти к творческому процессу более профессионально, и таким образом, книга обрела настоящий вид.

Также, в мои планы входит (в случае положительной оценки книги квалифицированными разработчиками и наличия у меня свободного времени) дальнейшее развитие теории «единого пространства УПР» в виде сайта в сети Интернет.

Сайт представлял бы собой информационный портал для обмена информацией между разработчиками ПО и пополнения БД УПР. Основные компоненты сайта – форум разработчиков, разбитый по темам (этапам ЖЦ,

методологиям, продуктам и прочее), гостевая книга, профили пользователей сайта и самое главное – БД УПР, доступную для публичного доступа. Каждое новое УПР (или изменение описания уже существующего) оценивалось бы пользователями сайта, и в случае положительного результата, добавлялось бы в БД.

Однако получилось то, что получилось. Что касается продолжения книги, то я думаю, что оно будет написано в тот момент, когда мои профессиональные познания выйдут на качественно новый уровень. Может быть, на это понадобиться лет двадцать (как у Брукса), а может раньше.

Кстати, по поводу Брукса. Вы заметили различия между посвящениями 1975 и 1995 годов? В первом упоминаются непосредственный и самый главный начальник. Во втором же - «Нэнси, Божьему дару для меня». Наконец-то в списке жизненных ценностей появляется семья и всё, что с этим связано. Не в этом ли заключается «сущность и акциденция программной инженерии»?

5. Благодарности

Итак, кто оказал на мое развитие наибольшее влияние и кому я за это благодарен:

- Мои родители – они меня родили ;-) Хотелось бы жить не во время победившего мирового терроризма, но ничего уж тут не поделаешь.
- Моя сестра – за поддержку.
- Технический Университет Молдовы – за получение не самого хорошего высшего технического образования, но лучшего из возможных в республике. Как сказано в книге [15] – «Поставь себе цель. Получи такое образование, какое только можешь, но затем, ради Бога, делай что-нибудь!».
- Стратан Виктор – за отличное объяснение в теории и практике паттернов проектирования, ООП и т.п. Первая версия Rational Rose 98, полученная мной – его заслуга.
- Сердцев Виталий – после наблюдения его шаманства с ассемблером, C++, низкоуровневого программирования, я стал «серьёзно» увлекаться собственной профессией и до сих пор об этом не жалею. Вспоминаю слова нашего завкафедры В. Бешлиу: «Вы пришли в Политех из разных мест и с разными целями, но если вы его закончите, то станете настоящими патриотами своей профессии».
- Смолов Александр – когда баланс между радостями и печалями профессии начинает сдвигаться в сторону последних (см. книгу [6]), я сажусь за компьютер и в течение месяца борюсь с пришельцами в старом добром XCOM

(см. сайт [15]), принесенных мне Александром на трёх дискетах со слониками. Грустные мысли от этого проходят, но потом еще долго снятся зеленые человечки ;-)

- Драгонер В.В. – куратор моего диплома. Была поставлена амбициозная цель по созданию аналога Rational Rose. Я написал компилятор исходных текстов C++; Сердцев – графическую оболочку для браузера объектов и диаграмм; Смолов - озвучивание на русском языке результатов анализа текстов через движок MSTextToSpeech. Проект благополучно провалился (к сдаче диплома было готово не более 50% заявленных требований), но неудачи иногда более полезны, чем успехи. Был получен опыт работы с Розой, прочтены материалы о Rational и многое другое.
- Папроцкий И.Б. и Старащук А.И. – работа в ГП «Registru» (см. статью [1]) под их руководством имела огромное значение на мой профессиональный и карьерный уровень.
- Золотухина Е.Б. – за потрясающий курс по системному анализу и разработке ИС.
- Дурлештяну Эдуард (компания BitGenerator) - вместе работать было тяжело, но очень интересно. К большому моему сожалению, тяжесть «перевесила» интерес.
- И, наконец, Топорец Игорь – мой непосредственный начальник на данный момент и настоящий профессионал своего дела. Многие поставленные цели мне удалось достичь быстрее благодаря нему. Очень надеюсь и на обратное (т.е. что я тоже помог достижению его целей).

6. Методологии разработки ПО

6.1. RUP

Rational Unified Process – Рациональный Унифицированный процесс.

RUP был создан в 1996г. корпорацией Rational при участии Гради Буча, Айвара Якобсона и Джима Румбаха.

Это поистине фундаментальная работа по описанию успешных методик создания ПО. Жизненные циклы ПО, потоки работ, роли, деятельности, артефакты, продукты, поддерживающие большую часть этапов ЖЦ. Не зря эту методику называют «тяжеловесной». Поддержка языка UML. RUP в качестве самостоятельной базы знаний по объему и значению может сравниться разве что с MSDN.

Основная идея RUP – максимально четко распределить работу каждого участника процесса разработки. Самая большая проблема, по моему мнению, заключается в том, что трудно (или даже невозможно) найти таких людей, которые будут делать только то, что им сказано. Как скоро им надоест заниматься одним и тем же? Не теряется ли чувство ответственности (и как следствие – удовлетворение от результата) за выполненную работу при жестком фиксировании участков работ? Служит ли наличие согласованных интерфейсов по обмену информацией гарантией качества и эффективности работы?

Продукты, поддерживающие методику – в первую очередь, это продукты самой компании Rational. Основной продукт Rose (используется практически на всех этапах разработки), SoDA (автодокументирование), RequisitePro (управление

требованиями), ClearQuest (запросы на изменения), ClearCase (версионность), Administrator (управление репозиторием проекта), WorkBrench (настройка корпоративных процессов), Quantify (тестирование скорости кода), Purify (определение утечек памяти), PureCoverage (тест охвата кода), Robot (автоматизированное тестирование), SiteLoad (нагрузочное тестирование), SiteCheck (проверка «мертвых» Web-ссылок). В настоящее время доступен RUP, интегрированный в среду разработки Microsoft .NET (под названием Rational XDE).

6.2. XP

Extreme Programming – Экстремальное программирование.

Рождением (а точнее датой «официальной» регистрации) можно считать 2001г., когда в США, штате Юта 17 сторонников «легковесных» процессов разработки выработали манифест с основными постулатами. Идеологами методики можно считать Кента Бека, Уорда Каннингема и Рона Джеффриса.

Основные принципы: тесная коммуникация, постоянное тестирование, минимум документации и максимум гибкости. Впрочем, лучше привести текст манифеста (см. книгу [4]):

«Мы открываем лучшие способы разработки ПО, создавая его сами и помогая другим. Благодаря этой работе мы стали ценить:

- **Индивидуумов и взаимодействия** выше процессов и инструментов
- **Работающее ПО** выше всеобъемлющей документации
- **Сотрудничество с заказчиками** выше согласований условий договора
- **Реагирование на изменения** выше соблюдения плана

Это означает, что, хотя элементы в правой части также имеют свою ценность, но больше мы ценим элементы, расположенные слева».

Довольно краткие и понятные формулировки, делающие доступными высказанные идеи самым широким массам. XP впервые озвучила некоторые

совершенно революционные принципы разработки. «Это вам не понадобится», «Ищите самое простое решение, которое может сработать», «Любые сидящие рядом два разработчика могут поменять всё что угодно в системе», «Заказчик в любой момент может изменить требования» и др.

Однако я уже высказывал свою критику по поводу XP. Большая часть претензий к XP снимается, вследствие более детального знакомства с ней. Можно считать, что последние проекты, в которых я участвовал, были «в духе XP».

Осталась следующая критика:

- Идея «нахождения представителя заказчика в одной комнате с программистами» по моему мнению, мягко говоря, является фантастическим пожеланием. Никто не спорит, что коммуникация с заказчиком жизненно необходима. Но решение проблемы скорее находится, во-первых, в сфере разработки стандартных форм документов по взаимодействию (ТЗ, ТП). Во-вторых, в более быстрых итерациях (нужно помочь заказчику сформулировать и откорректировать своё видение системы)
- Отрицание этапа «большого предварительного проектирования» допустимо только при разработке тривиальных систем (несложные сайты с уклоном в сторону дизайна, а не программирования, простейшие однопользовательские программы с минимумом бизнес-логики и т.п.). Цитируя одного из авторов: «Я рассматриваю здесь решения, полезные при разработке *сложных*

систем. Если приложение простое, зачем тратить на него своё время?». В сложном и интересном проекте, с «богатой» предметной областью было бы преступной халатностью не сформировать *в самом начале* каркас системы, основные принципы его функционирования. Конечно, «настоящие XP-шники» стоически воспримут информацию в середине проекта о том, что обнаружился прецедент, заставляющий переделать большую часть кода (или ещё хуже выбросить его). Но я считаю это совершенно недопустимым.

- Десятки userstory (бумажки с несколькими предложениями, характеризующими прецедент пользователя), оставшиеся после завершения проекта, не могут служить в качестве надежной документации. Получается, что XP нацелена на быструю *разработку* ПО, но не на его *сопровождение*. Приведу цитату из книги [4] по поводу неудачи проекта ЗС, выполненного посредством XP (расчет зарплаты для Chrysler – могу подтвердить, что зарплата при своей внешней простоте одна из самых трудных областей бухгалтерии, в которой «утонула» не одна команда программистов). «Когда ушло достаточно много сотрудников, незаписанные сведения о проекте и групповая память были утрачены». Я думаю, что апологеты XP переусердствовали с минимизацией документации. Что для студентов может выглядеть

привлекательным, серьезных разработчиков должно насторожить.

Должен поделиться собственным ощущением от «работы в стиле XP». В отличие от RUP (или любой другой методики с *фиксированием* результатов этапов, посредством тех. задания, тех. проекта и т.п.) XP дает ощущение неуверенности и анархии *в начале* проекта. Бизнес-требования и архитектура меняются так быстро, что, просидев пару дней дома можно потом не узнать структуру базовых классов (даже если ты сам её придумал).

Сразу начинаешь понимать положение XP о 40-часовой рабочей недели без переработок. Зачем до 22_00 трудиться в поте лица над модулем, который завтра может вообще не понадобиться?

В середине архитектура стабилизируется, *конец* же проекта характеризуется полной уверенностью в себе. Никакое из изменений требований, которое может высказать заказчик, не кажется ужасным. За время постоянных модификаций архитектуру *приходится* перерабатывать таким образом, чтобы изменения выполнялись максимально просто.

Программисты выступают в роли пользователей созданного дизайна программы – если он имеет дефекты, то систему трудно менять ещё на этапе разработки и это вызывает дискомфорт. В результате, дизайн *вынужденно* улучшается, а система легко переносит любые изменения бизнес-требований.

Продуктов, поддерживающих методику просто нет. И это, наверное, самое главное достоинство. Не станешь же, в самом деле, считать

«XP-продуктом» текстовый редактор или средство обеспечения версионности.

6.3. SADT

Structured Analysis and Design Technique - Методология структурного анализа и проектирования.

Известна как разработка компании SofTech, либо как только функциональный вариант в правительственной версии (IDEF0). Её начали применять с 1973г. во многих областях, таких как бизнес, производство, оборона, связь и организация проектирования.

Диаграммы в стандарте IDEF0 имеют несомненное преимущество для функционального моделирования системы. Однако представление модулей системы в виде блоков с набором входов и выходов и набором управляющих воздействий на них хорошо раскрывает только *высокоуровневые структурные особенности* системы. В этом SADT успешно конкурирует с диаграммами деятельности языка UML.

В SADT я не нашел механизмов для дальнейшей разработки системы, от сбора требований к реализации, тестированию и внедрению. Возможно, такая задача и не ставилась идеологами. Методика серьезно проигрывает остальным по охвату этапов ЖЦ ПО, сконцентрировавшись только на сборе требований и бизнес-моделировании.

SADT в полной мере реализует идею «большого предварительного проектирования в начале разработки» с целью уменьшить «просачивание» ошибок на более поздние этапы ЖЦ, где дороже их исправление (см. книгу [24]).

Из-за жесткой декомпозиции процесса разработки методику можно считать «прародителем» RUP.

6.4. MSF & MOF

Microsoft Solutions Framework (Набор решений от MS) и Microsoft Operations Framework (Набор операций от MS).

По замыслу составителей методики, она объединяет лучшие принципы каскадной и спиральной моделей разработки ПО, так сказать «лучшее из двух миров» (см. сайт [23]). Естественно, для достижения успешного результата предлагаются решения и продукты непосредственно от Microsoft.

Однако нельзя не признать огромную заслугу компании Microsoft в сфере компьютерной индустрии. Стоит изучить MSF только из-за того, что «так делает лидер».

Например, меня заинтересовало исследование, посвященное объединению ролей в малых проектах. Это особенно актуально для СНГ, т.к. большая часть ИТ-проектов, *декларируемых здесь* как большие и сложные, по меркам западных компаний являются мелкими и средними.

Очень хорошее описание процесса тестирования и сопровождения ПО. Кто из нас не ругался на полные ошибок RC-версии и на огромное количество всевозможных патчей, сервис-паков, хот и баг-фиксов, любезно предлагаемые для на сайте компании?

Толковая документация по практике оценки рисков при создании ПО. Ну и конечно, замечательные УПР в области продажи продуктов.

6.5. Iconix

Что-то среднее между очень громоздким RUP и весьма компактной XP (см. книгу [1]). Я считаю самым важным отличием от остальных методик введение диаграмм для анализа пригодности. Книга замечательная, начиная от TOP10 ошибок для каждого этапа разработки ПО и заканчивая постоянными напоминаниями вида «Над этим долго не работайте, сейчас у вас нет полной информации, чтобы тратить на это много времени».

7. Единое пространство решений

7.1. Этап ЖЦ «Управление»

7.1.1. Подбор команды

Наиболее полно освещающая эту тему, на мой взгляд, мне показалась книга [10]. Очень важно, что в ней рассмотрен процесс найма специалистов именно в области разработки ИС, хотя многое применимо и для других областей.

Также, мне показались довольно информативными (хотя и не без примеси саморекламы) работы на сайте [14].

Суть удачного решения заключается в том, чтобы набирать в команду людей, которые действительно заслуживают этого. Эти люди готовы брать на себя ответственность за принятие решений, играть различные роли в процессе разработки, они обладают значительной квалификацией в смежных областях разработки ПО. В отношении таких людей Джоэл Сполски употребляет слово «суперзвезды».

Спорное утверждение. Суперзвездами не рождаются, ими становятся. Все мы когда-то были начинающими программистами, поэтому я думаю, талантливые новички тоже имеют право быть в команде крупного проекта.

Что касается необходимости обучения новичков и не только их, а также проблемы «спаянности» команды из разнородных по квалификации специалистов, то здесь мне кажется, подойдет метод «маленькой победоносной войны». Он заключается в том, что команда (основные проектные решения, архитектура, инструменты разработки) сначала «обкатываются» на пилот-проекте, не имеющего критического значения для бизнеса. А после его успешного завершения, начинается уже основной проект.

7.1.2. Распределение ответственности

В больших компаниях чаще всего наблюдается стремление ужесточить контроль над принятием решений, работа разбивается на узкие участки с четко определенными полномочиями и ответственностью. Как правило, в начинающих компаниях, один и тот же человек выполняет несколько обязанностей, далеко выходящими за пределы его квалификации.

Можно привести цитату Александра II: «Россией управлять несложно, … но бессмысленно». Большие системы требуют большего управления. Однако, внутренняя самоорганизация, по моему мнению, эффективнее жесткого внешнего управления. В книге [6] приведен пример с компанией IBM, в которой более широкое делегирование полномочий от центра структурным подразделениям позволило кардинально увеличить управляемость всей компании.

Я думаю, что если не доверяешь людям, которые участвуют в проекте, то лучше вообще не начинать такой проект. Поэтому, лучше дать людям власти столько, сколько они смогут «унести». В конечном счете, если кто-то не справится с работой, его всегда можно уволить.

Когда каждый отвечает за свою узкий участок работы, это, по моему мнению, приводит к большим проблемам с качеством продукта и бессмысленной борьбе функциональных подразделений за перекладку ответственности между собой.

7.1.3. Атмосфера в проекте

Хорошие человеческие отношения между участниками проекта очень важны как для удачного завершения текущего проекта, так и для работы в будущем. В книге [4] говорится: «Удачная методология использовалась в проекте или нет, можно определить, если задать разработчикам вопрос – захотят ли они еще раз участвовать в аналогичном проекте или нет».

То же самое относится и к атмосфере проекта. Если рабочий день проходит в бесконечных разрешениях конфликтов; работники четко разделены на «касты» и не делятся информацией и знаниями с «противоположным лагерем»; нет благоприятных гигиенических факторов работы, наподобие отдельного телефона, звукоизоляции, удобного рабочего места, достойной оплаты труда и т.п., то вряд ли можно ожидать от работника, что он захочет испытать подобное еще раз (см. книгу [5]).

К сожалению, во многих случаях (большая текучесть кадров, недостаточные финансовые ресурсы и т.п.), работодатель не обращает внимания на стрессовую ситуацию в проекте, считая это нормой, и старается максимизировать прибыль путем сокращения издержек, связанных с налаживанием благоприятной атмосферы.

Я считаю это ошибкой. Необходимо использовать любую возможность для повышения производительности труда, начиная от бесплатной пиццы за внеурочную работу и заканчивая персональными абонентами в фитнес зал для поддержки спортивной формы сотрудников.

7.1.4. Карьерный рост

Человек не является роботом, механизмом, который может всю жизнь крутить одну и ту же гайку на одном и том же станке, прерываясь на редкие капремонты (отпуска) и периодическое смазывание машинным маслом (зарплата в конце месяца). Большинству из нас хочется добиться чего-то большего, чем обладаешь в данный момент.

Поэтому, я считаю, что нужно ставить перед собой реальные цели и каждую минуту работать над её достижением. К сожалению, постоянное отсутствие свободного времени приводит к тому, что главным (а зачастую и единственным) местом для повышения своей квалификации и развития является рабочее место.

Работодатель должен (несмотря на кажущуюся убыточность вложений на «быстрый» Интернет, новые книги, курсы и даже поездки на конференции типа PDC – см. статью [2]) обеспечить рост (профессиональный, карьерный) своих сотрудников. В конечном счете, все остаются в выигрыше – сотрудник становится более удовлетворенным работой, а организация получает более профессиональные кадры.

Конечно, возрастает риск потерять работника, ставшего «более квалифицированным», чем требуют его должностные обязанности. Как привлечь и удержать талантливых сотрудников, см. книгу [10].

Я согласен с мыслью, высказанной в той же статье [2]: «НЕ РАБОТАЙТЕ С НАЧАЛЬНИКОМ, КОТОРЫЙ АКТИВНО ПРЕПЯТСТВУЕТ ВАШЕМУ ПОСТОЯННОМУ ОБУЧЕНИЮ. НИ В КОЕМ СЛУЧАЕ».

7.1.5. Производительность труда

На эффективность работы сотрудника влияют множество факторов. Физические факторы - величина личного пространства, освещенность, звукоизоляция, размер и качество монитора. И чисто «виртуальные» - атмосфера в проекте, взаимоотношения в коллективе, языковая среда и культурные различия. Как бы то ни было, я считаю, необходимо выявить всевозможные источники дискомфорта, мешающие нормальной работе и устранить их.

Намного выгоднее, решить возникшую проблему еще в зародыше, когда она еще не привела к явному конфликту. Люди – слишком ценный ресурс, чтобы их терять в процессе неуклюжих действий по решению проблемы.

7.1.6. Коммуникация

Очень многое в проекте зависит от скорости и способов передачи информации. Конечно, хотелось бы (как того требует методология XP) наличия единой комнаты (deathmatch room), в которой бы бок о бок находились бы все разработчики, совместно с представителем заказчика, и никто кроме них. Однако, в большинстве случаев это невозможно (не собираться же, в самом деле, всем разработчикам в актовом зале компании).

Поэтому, необходимо использовать максимально эффективную из доступных средств коммуникации.

Для больших бюрократических (чаще государственных) организаций подойдут документы в бумажном виде, передающиеся от одного согласующего лица другому. Для малых ISV (Independent Software Vendor – независимый разработчик ПО) ничего лучше, чем совместная работа разработчиков «на расстоянии вытянутой руки» пока не придумано. Компании, имеющие в наличии «распределённые» географически команды, используют видеоконференции и различные средства взаимодействия через Web.

Кроме эффективной коммуникации необходимо применять и её отсутствие, т.е. противодействовать так называемым «информационным сквознякам» - см. книгу [4]. Нет ничего хуже, чем во время разработки слышать (видеть и т.п.) постороннюю информацию, не имеющую отношения к текущему проекту (например, сидеть в одной комнате со служащими отдела сопровождения, продаж или рекламы).

7.1.7. Планирование

Первое правило планирования – используйте его, каким бы бессмысленным оно вам не казалось. У большинства разработчиков ПО сложился весьма заметный пессимизм по отношению к планам (и совсем пренебрежительное отношение к людям, занимающимся планированием). Программирование – это не строительство, где точно известно, что кирпичную стену размером 5м^2 строитель сложит за час, а раствор марки Х застывает за 10 минут.

Слишком много взаимосвязанных факторов нелинейно влияющих на общий результат. У меня, например, получилась такая формула: задача занимает в 2-3 раза больше времени (ближе к 2-м), чем рассчитываешь на неё потратить, в случае, если всё пойдет как надо. Т.е. найдутся все нужные компоненты, быстро будет придуман алгоритм, не будет ошибок в используемых модулях и, наконец, самое главное – удастся сконцентрироваться на задаче и не отвлекаться.

Цель планирования – «Знать, на сколько проект отстаёт и вовремя перераспределить задачи, чтобы пусть и с меньшей функциональностью, но уложиться в график» – см. книгу [16]

Большие и детально проработанные планы, как правило, проваливаются – слишком велика погрешность в каждой компоненте графика. Поэтому, я считаю рациональным практику «маленьких победоносных войн» (итеративная разработка), когда определяются наиболее приоритетные задачи, выполнимые в пределах 1-4-х недель и команда планирует и старается выполнить их в заявленные *малые* сроки.

7.1.8. Организация процесса

Конечно, для стандартизации процесса разработки ПО уместно создание методики – перечня правил, фиксирующего основополагающие моменты всех этапов жизненного цикла программного продукта. Сам факт того, что в организации используют такую методику (или пытаются её создать) говорит о высокой степени зрелости ИТ-компании.

Но во многих случаях, методика является «рекламным трюком» компании, на бумаге прописывающей хорошие и здравые идеи, на самом деле далекие от действительности. Таким образом, компания пытается получить конкурентное преимущество в борьбе за получение выгодных заказов. В тот момент, когда выясняется беспомощность организации в вопросах организации работ, оказывается слишком поздно.

Еще хуже, когда чересчур жесткая фиксация процесса разработки мешает эффективной разработке проекта, «не влезающего» в рамки, очерченные методикой. Как правило, каждый проект уникален, поэтому я считаю рациональным использовать набор УПР, и гибко их использовать в той или иной ситуации.

Как можно понять из книги [4], «главное – отрубить противнику руку, а не воспользоваться определенным боевым стилем».

Главная черта успешной ИТ-компании, занимающейся коммерческой разработкой ПО, по моему мнению, является гибкость в вопросах «настройки» под различные типы клиентов (проектов, предметных областей), а не сертификаты на соответствие «N-го» уровня CMM (см статью [8]).

7.1.9. Функции разработчиков

Хорошая армия нуждается во многих родах войск: пехоте, авиации, разведке, морском флоте и многих других. Кроме *наличия* видов вооружений, основная цель состоит в том, чтобы наладить их эффективное и выгодное *взаимодействие*.

Я не буду с этим спорить. Но я считаю, что есть одно важное различие между ведением военных действий (и их успешным завершением) и разработкой программных проектов. Оно заключается в том, что люди на войне (по мнению штаба) взаимозаменяемы. Т.е. ради захвата района X, можно пустить на «пушечное мясо» дивизию Y, а потом провести мобилизацию и заново создать дивизию Y".

В ИТ-проектах всё иначе. Каждый разработчик обладает уникальной информацией, и, потеряв одного из них, нельзя сразу же нанять другого, не потеряв в качестве.

Существует такой вопрос: «Сколько разработчиков должен задавить грузовик, чтобы проект провалился? Худший ответ - одного». То есть речь идет не только об уникальности каждого сотрудника, но и об устойчивости команды в случае потери одного из них.

Исходя из вышеизложенного, я прихожу к следующим выводам. Во-первых, важно помнить о ценности каждого сотрудника. Во-вторых, еще более важно, обеспечить их взаимозаменяемость. Достижение второй цели, на сегодняшний момент, я вижу только в обеспечении разностороннего профессионального роста и ротации сотрудников в различных функциональных подразделениях.

7.1.10. Обучение персонала

Новый человек, пришедший в проект, возможно будущая звезда компании. Поэтому не нужно жалеть времени на интеграцию его в коллектив, обеспечение всеми необходимыми материалами и отрывать ведущих разработчиков на общение с новичком и его обучение.

«Новый человек» в начале своей работы нуждается в благоприятном окружении и чувстве связности своей работы с работой остальной компании. Этого можно добиться путем участия новичка в обсуждениях текущих проектов (даже не имеющих отношения к основной работе, поставленной перед сотрудником). Главное – добиться ощущения единой команды и её усиления посредством использования знаний, опыта и вообще эффекта «свежей головы» новичка.

Вопрос о «новых людях» в проекте тесно связан с другим вопросом: «Когда их нужно и выгодно принимать в проект?». Как сказано в книге [6] добавление людей в проект, находящийся на стадии завершения только удлинит срок реализации, т.к. много времени будет потрачено на введение новичков в контекст, информационное окружение рабочего проекта».

Иногда руководство видит единственное решение в «спасении проекта» только в добавлении новых людских ресурсов. В таких случаях, я считаю, необходимо проанализировать, на что тратят время ключевые разработчики, и поставить перед ними только приоритетные задачи (доработку проекта), а второстепенные задачи передать на исполнение другому персоналу, в т.ч. новичкам.

7.1.11. Ориентация на задачи

В книге [4] приводился пример с тестировщиками, которых сильно беспокоило отсутствие видимого прогресса в работе программистов. Чтобы минимизировать отвлечение программистов от работы, было принято решение вывесить список задач в проекте и отметки об их выполнении в коридоре, доступном для всеобщего обозрения. Вопросы сразу прекратились.

Данный пример демонстрирует, как важно для морального духа команды иметь «перед глазами» информацию о текущем прогрессе каждого элемента проекта. Данная техника оказывает самое благотворное влияние на разработчиков, в связи с ощущением того, что «прошел ещё один день, и зачеркнуто еще X пунктов». Каждому захочется выделиться, чтобы сделать больше, чем остальные.

При этом не важно, какими инструментами это будет достигнуто: планом в MS Project, списком задач в MS SharePoint или просто файлом MS Word с нумерованным списком задач и местом для галочки.

Следует отметить ещё один чисто психологический момент. Ничего не убивает так быстро активность и оптимизм команды, как отсутствие ясной и *реально достижимой* цели в заданные сроки.

Поэтому, скорее всего можно представить разработчика, который знает, что до выходных ему нужно сделать функции X, Y и Z и который отказывается от обеда, чтобы успеть до этого срока. Нежели того же сотрудника, *активно* работающего в отсутствие четкого представления о прогрессе команды и доли своего участия в нём.

7.1.12. Общая среда проекта

В книге [6] говорится о «Рабочей тетради проекта в бумажном варианте с последующим переходом на микрофиши». Сейчас это выглядит смешно, но рациональное зерно состоит в том, чтобы иметь некий согласованный набор документов, описывающих как концептуальные основы проекта, так и ход выполнения работ.

Вопрос не только в составлении такого набора документов, но и в том, чтобы все участники проекта имели доступ к этой информации.

В той же книге [6] забавно было наблюдать, как автор полностью поменял свою точку зрения «в отношении сокрытия информации» под влиянием идей Дэвида Парнаса.

Возможно, когда-нибудь и я изменю свою точку зрения, однако на данный момент я считаю, что вся информация о проекте должна быть доступна для всех участников разработки.

И вот почему:

- Яркие технические решения в разных частях системы провоцируют их обсуждение разными людьми и как следствие – повышается их качество
- Заставив «гуру» выложить свои идеи на бумаге, увеличивается скорость обучения низко-квалифицированных сотрудников
- Четкое разделение потоков информации усиливает изоляцию групп (аналитиков, программистов, тестеров) друг от друга
- И, наконец, лучше иметь обилие информации (с возможностью выбора наиболее полезной на данный момент), чем её недостаток.

7.1.13. Интенсивность работы

Совет будет несколько смешной, но действенный: «Дозируйте свою энергию, больше отдыхайте». В книге [4] приведён почти анекдотический случай. Во время работы автора в центральном банке Норвегии выяснилось, что рабочий день длится только до 15^{30} (все уходят домой и общаться просто не с кем). В 17^{00} отключается освещение, кроме аварийного, а в 19^{00} вообще всё замирает. Так начальство заботится о том, чтобы сотрудники не перетрудились на рабочих местах.

Однако, смысл в том, что постоянное перенапряжение вредно. Лучше каждый день быть производительным на все 100%, чем в понедельник работать до 22^{00}, а в течение остальной недели проявлять активность дохлой рыбы на солнцепеке.

Многие проблемы решаются быстрее (во всяком случае, не кажутся такими страшными), если предварительно хорошо отдохнуть.

Из личных ощущений. Я знаю, что на данный момент я обладаю некоторой квалификацией. Я уверен, что завтра (после прочтения каких-то книг, статей, форумов в Интернете) моя квалификация повысится. Поэтому иногда, вместо настойчивости в желании «выловить последний баг», лучше отложить проблему. На следующий день она решится гораздо проще.

Это не отказ в упорстве достижения целей, а твердая уверенность в том, что всё намного проще, чем кажется на первый взгляд. Если задача решается трудно, долго и мучительно, есть большая вероятность, что выбрано неоптимальное решение.

7.1.14. Система приоритетов

Выставление приоритетов выполняемым задачам нужно для того, чтобы в каждый момент времени делать то, что обладает максимальной ценностью для заказчика продукта. Возникает очень неприятное ощущение, если после долгого труда над отдельной функцией, выясняется, что она редко используется клиентом или вообще не нужна. Поэтому очень важен приоритет требований и своевременное их изменение заказчиком в процессе развития системы (и бизнеса).

Заказчик имеет право изменять значимость отдельных элементов системы. Дело разработчиков – не противоречить ему, мотивируя это тем, что «функция X модуля Y уже наполовину разработана, и она вам точно понравиться», а быстро отреагировать на изменившуюся ситуацию.

В этом ключе привлекательна методика XP, предполагающая наличие заказчика «в одной комнате с программистами» и наиболее быстро реагирующая на изменение его требований.

Однако я всегда с большим скепсисом относился к такому стилю взаимодействия с заказчиком. Заказчик в большинстве случаев слабо осведомлен о возможностях, предоставляемых программной инженерией. Поэтому, основная черта успешной разработки, по моему мнению, – это не общение по телефону в стиле «хорошо, ваша новая задумка будет реализована в первую очередь, а написание остальных приостановлено», а как можно более раннее выявление возможностей системы, стратегически важных для бизнеса.

7.1.15. Документация

Речь идёт не о *доступности* информации в проекте (это УПР уже обсуждалось), а о *составлении* открытой и качественной проектной документации вообще (см. статью [3]).

Существует распространённое мнение, согласно которому «настоящий программист» не использует (разрабатывает, читает и т.п.) никакую документацию, тем более диаграммы (см. книги по XP), а комментарии кода вообще издают неприятные запахи (сказано было другое, но подразумевалось это – см. книгу [14]).

Я думаю, это неверно. Никакая бытовая техника не продаётся без инструкции, а чертежи космического корабля стоят дороже, чем его физическое воплощение.

Может дело в моей лени или плохой памяти, но мне действительно нравиться фиксировать принятые соглашения по архитектуре или особенностям алгоритма. Это позволяет быстро вникнуть в конкретные детали реализации, без её кропотливого анализа (лень) или быстро вспомнить о принятых решениях после бурно проведённых выходных (память).

Наличие качественной документации – гарантия устойчивости проекта (не будет мест, понятных только одному разработчику). Она дисциплинирует как этапы разработки (чтобы выложить мысли на бумаге, придется «причесать» их в понятный всем вид), так и облегчает процесс сопровождения (работники службы поддержки – чаще всего не первоначальные разработчики, и без документации им будет совсем плохо).

7.2. Этап ЖЦ «Анализ»

7.2.1. Представление информации

Одна из самых больших проблем, при взаимодействии с представителями «разных уровней понимания» системы (например, между заказчиком и аналитиком, аналитиком и программистами и т.п.), заключается в трудности «разговора на одном языке». Каждый мыслит в категориях, понятных «его группе». Заказчик – в терминах бизнес-процесса, программист – в понятиях классов, форм и SQL-запросов, тестеров же интересуют четко ограниченные функциональные требования с перечнем граничных значений для тестирования.

Я считаю очень важным как можно быстрее добиться форм представления информации, удовлетворяющей всем сторонам. Не нужно загонять себя (и противоположную сторону) в жесткие рамки определенных видов документов. Для «большого» совещания подойдет красочная презентация, для первоначального изучения предметной области – диаграммы бизнес-процесса, для детализации требований – текстовое описание прецедентов «по Коберну».

7.2.2. Стратегия продвижения

Я считаю более эффективной стратегию разработки предметной области, когда сначала как можно быстрее реализуется движение «вширь», а потом (после подтверждения того, что был выбран правильный путь или соответствующей корректировки курса) движения «вглубь».

Такой способ продвижения предупреждает разработчиков от известной проблемы «паралича анализа», связанной с невозможностью решить локальную проблему, из-за которой прекращается движение вперед.

После прочтения книги [1] я пришел к выводу, что практически в каждой главе дается совет вида «не тратить много времени на этап разработки X, переходите к этапу X+1». Основным аргументом такой поспешности действий является то, что на любом этапе доступно слишком мало информации, чтобы тратить большое количество ресурсов на принятие долгосрочных решений.

Необходимо «ухватить» основной сценарий взаимодействия с системой, а не погружаться в нескончаемые дебри запутанной бизнес-логики. Всё равно её окажется больше, чем будет описано. Многое выясниться только на этапе тестирования или уже после внедрения системы.

ИС создается достаточно долго, поэтому множество требований заказчика устареет еще на этапе разработки. Следовательно, лучше как можно быстрее реализовать «неполное, временное решение», чем в самом конце разработки дать заказчику «полное решение». У Фаулера то же самое сказано насчет тестов: «лучше выполнить неполные тесты, чем не выполнить полные тесты».

7.2.3. Две точки зрения

Как уже было сказано, «одна голова хорошо, а две лучше». При анализе предметной области очень эффективно иметь двух аналитиков, работающих в паре, чем каждого по отдельности. Связано это с тем, что, как правило, общение с заказчиком проходит в режиме диалога, поэтому всегда лучше иметь одного человека, который задает вопросы и ведет беседу, и второго, который записывает высказанные мысли.

Также, парная работа способствует обсуждению ключевых моментов с целью раскрытия их сути, а не просто фиксации того, что было высказано заказчиком.

7.2.4. Глоссарий терминов

Почему не удалось закончить строительство Вавилонской башни? Трудность заключалась в том, что внезапно участники разработки стали говорить на разных языках. Проблемы в отсутствии единой терминологии и одного языка общения, понятного всем: заказчикам, архитекторам и простым рабочим привели к провалу проекта.

Аналогичная ситуация в разработке ПО. Много недоразумений, которых можно было бы избежать, возникают из-за того, что одни и те же понятия называются по-разному. Заказчик оперирует понятиями бизнеса, проектировщики – мыслят категориями классов и паттернов, разработчики БД – таблицами и запросами.

Я считаю очень важным:

- Во-первых – выработать единую систему терминов и обозначений, согласованную всеми участниками разработки;
- Во-вторых - «пронести» понятия из предметной области в архитектуру системы.

Может оказаться хорошим тестом – показать заказчику текст функции, представляющей элемент бизнес-процесса. Если текст будет нашпигован адресной арифметикой, вызовом методов с именами типа GetDById(int i), то наверное, с кодом не все в порядке. Ту же функцию, как мне кажется, лучше было назвать GetDiscount(int DiscountCardId).

Правильно выбранные имена, сокращают необходимость комментариев и увеличивают понятность кода (что напрямую влияет на легкость дальнейшей модификации и сопровождения).

7.2.5. Диаграммы

Многие критикуют использование диаграмм из-за предполагаемой трудоемкости их составления и актуализации. Однако я считаю целесообразным любой способ эффективного описания элементов системы и способов их взаимодействия.

На данный момент наиболее известны следующие типы диаграмм (см. сайт [9], книгу [2]):

- Class (классов) – иерархии классов
- Object (объектов) – взаимодействия объектов в run-time, связи и отношения между ними
- Use-case (вариантов использования) – отношения между актерами и прецедентами
- Sequence (последовательности) – взаимодействия между объектами классов посредством обмена сообщениями
- Collaboration (взаимодействия) – аналогична Sequence, но упор на связях между объектами
- Statechart (состояний) – состояния сущности и способы перехода между ними
- Activity (деятельности) – бизнес-процессы, функциональная декомпозиция системы
- Component (компонентов) – отношения между составными элементами системы
- Deployment (размещения) – физическое размещение модулей

В книге [1] описываются еще диаграммы анализа пригодности для проверки на непротиворечивость логики описания прецедентов.

Также я считаю, выгодно разрабатывать свои типы диаграмм (или смешивать существующие), для более точного описания системы. В любом случае, важна не «идеологическая чистота» элементов диаграммы, а получение пользы от нее.

7.2.6. CASE-инструменты

Хороший инструмент для рисования диаграмм, сбора требований, построения моделей предметной области значительно облегчает работу системного аналитика. Существует мнение, согласно которому успех методики RUP (Rational Unified Process) в значительной степени обусловлен наличием качественных инструментов компании Rational, поддерживающих все этапы ЖЦ продукта.

Преимущества и недостатки CASE-инструментов вызывают многочисленные бурные дискуссии. Самым важным вопросом в обсуждении того или иного инструмента, я считаю его гибкость и «идеологизированность». В статье [4] Rational Rose подвергается жестокой критике, за чересчур вольную трактовку языка UML.

В своё время я высказывал резко негативное отношение к этой статье (из-за критики Rose). Также я был противником превращения Rose в «графический редактор для рисования диаграмм».

Сейчас моё мнение изменилось. На данный момент, я считаю, что чем функционально богаче, проще и легче в понимании продукт, тем лучше. Низкие требования к знанию того или иного языка моделирования, необходимые для начала использования продукта, уменьшают время обучения новичков и тем самым увеличивают популярность продукта.

7.2.7. Прецеденты

Прецедент (П-т) (или "user story" – пользовательская история в методике XP) представляет собой краткое, лаконичное описание отдельной функции бизнес-процесса, имеющей ценность для заказчика.

Под «имеющей ценность» понимается, что клиент готов платить отдельную плату за реализацию данного конкретного П-а. То есть, разработка нового класса, создание механизма доступа к БД – это не П-т. Заказчику всё равно, какова будет реализация его бизнес-потребностей.

Авторизация при старте системы, создание нового заказа, оплата покупки, получение отчетности – это «нормальные П-ы». Каждый из них может быть реализован отдельно и «продан» заказчику за плату. По мере приобретения созданных П-в, заказчик получает всё большую выгоду от разрабатываемой ИС.

П-т является воплощением принципа «разделяй и властвуй». После разделения ИС на мелкие, чётко определенные подзадачи, становится значительно проще их описывать, реализовывать и оценивать прогресс их выполнения. Риск возникновения ситуации вида «90% модуля готово…, и еще 90% осталось…» значительно снижается.

Хотелось бы отметить замечательное описание теории и практики составления, структурирования и использования П-в, представленное в книге [3]. Ещё раз повторю основные преимущества текстовых прецедентов: а) текст проще составлять; б) понятнее заказчику; в) легче обучить их составлению.

7.2.8. Реинженеринг бизнес-процессов

Этот термин стал очень популярен за последнее время. Означает он, прежде всего, переосмысление, перестройку, повышение эффективности бизнеса и процессов, его поддерживающих. Зачастую, необходимость реинженеринга диктуется созданием новой ИС, автоматизирующей деятельность организации.

К сожалению, во многих случаях, автоматизация, особенно в крупных организациях, служит лишь отправной точкой для начала «большой войны за передел сфер влияния» между структурными подразделениями и их руководителями. Особенно плохо, что в ситуации, когда проигрывает менеджмент, в первую очередь страдает команда разработчиков.

Во многих случаях, когда ставится задача не просто построения новой системы «с нуля», а изменения существующих бизнес-процессов, полезно построение моделей системы AS IS (как есть) и TO BE (как будет), с целью выяснения различий (и своевременного их согласования).

Всегда полезно перед «автоматизацией хаоса» (см. статью [5]) сначала выполнить упорядочивание процессов. Это делает ненужным написание огромного количества лишнего кода, что в значительной мере экономит время и силы проектной команды.

7.3. Этап ЖЦ «Проектирование»

7.3.1. Создание объектов

В книге [8] описаны такие характеристики качества классов и объектов, как: зацепление, связность, достаточность, полнота и примитивность. Каждая придуманная абстракция должна выполнять определенную работу, используя ограниченный набор информации. Если для получения результата объекту чего-то не хватает, он «просит» объекты других классов помочь ему в разрешении проблемы.

Создавать «правильные» объекты предметной области, эффективно взаимодействующие между собой – большое искусство. В проекте расчета зарплаты мне было подсказано элегантное решение о методе расчета суммы надбавок. Оно состояло в том, что каждая надбавка «знает» как вычислить свою сумму, либо может «попросить» у своих компонентов передать ей их суммы и т.д.

Таким образом, каждый объект решает только ту задачу, для решения которой у него есть все доступные ресурсы. Так в задаче, которая была бы выполнена в чисто «структурном» стиле (циклы, рекурсия) с помощью ООП было реализовано удивительно красивое решение.

Тем же человеком (см. последний пункт главы «Благодарности») было высказано весьма самоуверенное мнение: «Код каждого метода должен быть не более трех строк, либо одна строка в виде делегирования обработки методу другого объекта». Смысл на самом деле не в числе строк, а в простоте кода – в хорошо спроектированной системе очень мало кода и очень много взаимодействий объектов между собой.

7.3.2. Паттерны проектирования

Замечательное описание паттернов проектирования (ПП)– находится в книгах [7, 17 20, 9 и 19]. Как сказано в книге [14]: «Сейчас уже невозможно делать вид, будто понимаешь что-то в объектах, если не умеешь с умным видом говорить о стратегиях, одиночках и цепочке ответственности».

ПП не учат чему-то принципиально новому, скорее они стандартизируют уже имеющиеся УПР в области проектирования каркаса ИС и написания «правильного кода».

Квалифицированный разработчик, хорошо разбирающийся в концепциях ООП, умеющий мыслить абстрактно и строить красивую и гибкую архитектуру, и так применяет ПП, пусть и неосознанно для себя. Начинающему же программисту они практически бесполезны – вряд ли абстрактная фабрика или Memento понадобятся в тривиальных проектах.

Скорее всего, ПП нужны среднему разработчику, повышающему свою квалификацию, имеющему достаточный опыт программирования «на коленках» и понимающему, что настало время кардинально повысить свой уровень образования.

Еще два положительных момента в ПП. Во-первых, они вводят единый словарь общения - достаточно сказать: «Сделай этот объект синглетоном», как каждому станет понятно, о чем идет речь. Во-вторых, на порядок облегчается доработка системы, если известно, что к объектам были применены определенные ПП (иначе, ПП и прочие решения «растворяются в коде» и сопровождение на те же порядки усложняется).

7.3.3. Компонентная разработка

Концепция разработки систем с помощью изолированных модулей, выполняющих требуемую функциональность по определенному интерфейсу, давно завоевала популярность. Компоненты значительно расширили круг разработчиков ПО и снизили требования к их квалификации.

Существует мнение, что успех современного программиста в значительной мере определяется его способностями по использованию компонентов. К сожалению, у компонентов есть один недостаток, обусловленный их преимуществами. Они не заставляют думать в терминах объектах и модели предметной области. Шутка насчет того, что «при возникновении проблемы, программист Делфи начинает бомбардировать Web-конференции вопросами про «магический компонент», преодолевающий возникшие трудности, а если его не обнаруживается, объявляет, что проблема неразрешима», всё больше становится правдой.

Дело вовсе не в критике Делфи и RAD-систем визуального программирования (я использую среду .NET которая как раз из этой серии). Просто это распространенный стереотип: начинающий программист, научившийся «кидать» компоненты на форму и объявляющий себя мастером ООП (кнопки, что ни говори, всё-таки экземпляры классов).

Большим достижением компонентов является возможность параллельной работы над отдельными частями системы. Купленные компоненты способны значительно ускорить разработку. Вряд ли кому-то понравиться раз за разом приделывать к Grid сортировку, вычисления суммирующих значений и прочие «стандартные» возможности.

7.3.4. Концептуальная целостность

Целостность системы, заранее продуманные и реализованные механизмы реакции программы на возникающие изменения требований, по моему мнению, ключевой фактор успеха не столько разработки, сколько сопровождения и развития ИС. Не может быть ничего хуже, чем обнаружить, что новое требование заказчика полностью разрушает «идеологию» системы и мы были не готовы к изменениям такого рода.

Это не аргументы в защиту метода «большого предварительного проектирования и фиксации требований» ИС посредством технического задания. Наоборот, я прихожу к выводу, что более рационально применение коротких и быстрых итераций через прецеденты при начальном *грамотном* построении *каркаса* ИС. Однако архитектура должна быть *гибкой* и устойчивой к модификациям. Точного ответа на вопрос, *как именно* получить такое решение, сейчас я не знаю.

Тем не менее, я уверен, что оперирование концепциями объектов, взятых непосредственно из предметной области, значительно повышает шансы на успех проекта.

Задача выработки основных принципов функционирования ИС и слежения за «идеологической чистотой» применяемых решений, как правило, находится в ответственности архитектора системы. Главное отличие архитектора от менеджера проекта (их часто путают) состоит в том, что архитектор – это технический, а менеджер проекта – административный лидеры процесса разработки. Потеря того или другого может иметь катастрофические последствия для создаваемой ИС.

7.3.5. Распределение ошибок

Как сказано в книге [14] необходимо добиваться «сдвига» ошибок со времени выполнения (run-time) на этап компиляции приложения. Идея о том, что программу нужно строить так, чтобы ошибки сигнализировали о себе в момент разработки, когда их относительно легко исправить, а не во время выполнения программы.

Использование адресной арифметики, операций вида sizeof(ObjectType) где делаются предположения о размере объектов, является впечатляющим примером того, как гибкость языка (например, C++) значительно повышает вероятность возникновения трудноуловимых ошибок, «видимых» только в run-time.

Статья, где сказано о более быстром доступе DataReader-а к полю таблице БД по индексу, а не через название, как мне кажется, также внесла свою «негативную лепту» в процесс построения ПО.

Так теоретический тезис о нарушении инкапсуляции (внутреннего устройства объекта) приводит к вполне практическим ошибкам.

Неявные преобразования типов (например, из int в bool), предположения о значении некоторых констант (true=1, а false=0…, или -1?), «явное» управление получением и освобождением памяти в противовес автоматической сборке мусора – всё это способы неправильного распределения ошибок.

Наоборот, строгая типизация типов, введение интерфейсов (класс обязан реализовать строго определенное подмножество методов), упрощение иерархии наследования (любое приведение типа от суперкласса к наследнику, потенциально опасно), помогут «правильно» распределить ошибки.

7.3.6. «Неправильные» решения

Один мой знакомый высказал негодование по поводу того, что конструктор, который он купил своему ребенку «неправильный» - нет нужных деталей, в положенных местах не просверлены дырки и прочие недоработки. Я высказал предположение, что таким образом фирма-производитель внушает ребенку мысль о несовершенстве мира. А чтобы он стал лучше, его нужно «слегка» обработать напильником.

Цитируя того же знакомого: «Если система местами «ненормальна» - то это лишь признак того, что она реально работает, а не создана в академических интересах».

Таким образом, не стоит пугаться того, что ваша БД не в 5-ой НФ, в таблицах сохраняются вычисляемые поля, а для хранения продажи записываются названия товаров, а не их Id.

Часто обилие связей между объектами, «красиво» выглядевшие на диаграмме классов (из-за чего один объект может получить все данные второго, а через него третьего и т.д.) приводит к большим проблемам с производительностью. В данном случае «спасают» ленивые коллекции (загрузка по требованию), пул выданных объектов, кэширование и т.п., но это варианты *решения* проблемы, а не её *предотвращения*.

В любом случае, лучше спокойно воспринимать мысль о том, что иногда нужно делать вещи, противоречащие фундаментальным принципам построения ИС. В конечном счете, главное – это удовлетворение заказчика, а не детальное соблюдение «идейных принципов».

7.3.7. Изобретение колеса

Как это ни горько сознавать, но программирование всё больше превращается в ремесло, нежели в искусство. Для того чтобы успешно создавать программные продукты, вполне достаточно качественно реализовать существующие концепции, а не придумать новые.

Поэтому, важно рационально использовать ресурсы в проекте – не вносить в систему излишнюю сложность, делая вещи, уже реализованные кем-то другим. При любой возможности старайтесь превратить «большой» проект в «малый» за счет упрощения принимаемой архитектуры и ограничения множества реализуемых функций (см. книгу [19]).

Конечно, соблазнительно в ходе разработки получить опыт создания подсистемы, дублирующей уже известную функциональность операционной системы, СУБД или графических редакторов, но вот вопрос – готов ли за это заплатить заказчик?

Творческую активность необходимо применять в слабоизученных областях, которых становится всё меньше. В большинстве случаев, стоит положиться на стандартные решения, давно существующие и опробованные в отрасли.

Я считаю весьма эффективным повторное использование – идей, опыта, кода, компонент, приложений и т.п. Чаще всего, собственноручное изготовление компонент (а не купленных), создание механизма бизнес-транзакций (против системных в СУБД) и т.п. – прямой путь к значительному увеличению сложности проекта и увеличению сроков реализации системы.

7.3.8. Алгоритм

Изначально «быстрый» алгоритм, использующийся в приложении, значительно снижает как потребность в повышении производительности в дальнейшем, так и стоимость такой оптимизации. Многие испытывали на себе «магическое» воздействие учебного примера из среды Borland Delphi/Builder, где в графическом виде показывалось преимущество одного из трёх различных видов сортировки.

Однако, как сказано в книге [19] «с возрастанием степени параллелизма операций в корпоративных приложениях разработчики все меньше заботятся о сопутствующих проблемах». Аналогичная ситуация с алгоритмами. С появлением развитых библиотек функций, большинство программистов забыли, чем пузырьковая сортировка отличается от сортировки вставками.

Наверное, оно и к лучшему. Можно сэкономить время на чтении таких книг, как [21, 22, 23]. Использование библиотечных функций уменьшает вероятность ошибки прикладного программиста, а код становится «чище» и понятнее.

Иногда всё же без самостоятельного создания алгоритма не обойтись. В том же проекте расчета зарплаты был изменен первоначальный алгоритм нахождения надбавок, которые *не зависят* от уже имеющихся у служащего. Задача наводила мысли о рекурсии и бесконечных циклах. Было предложено определять те надбавки, которые *зависят* от уже имеющихся (это довольно легко), а потом через разность с общим множеством надбавок получать требуемый результат. Это и было сделано, что дало огромный прирост производительности.

7.3.9. Расслоение системы

Принцип Model-View-Controller (MVC – Модель-Представление-Контроллер) давно известен, и обсуждается, например, в книге [19] и статье [6]. Суть метода заключается в «расслоении» системы по трем логическим уровням.

View – презентационный уровень. Включает в себя формы пользовательского интерфейса или их аналоги. Это то, что «видит» юзер и с чем он взаимодействует, пытаясь получить полезное действие у системы.

Model – работа с данными, СУБД, запросы, хранимые процедуры и прочие «внутренности» организации данных системы. Пользователь не должен «напрямую» видеть внутреннее хранилище информации (этот принцип так часто нарушается, что я даже не буду агитировать за него).

Controller – бизнес-логика, классы модели предметной области, сервера приложений, сервисы, службы, свои механизмы для обработки данных (пул выданных объектов, уведомления о событиях, ленивая загрузка и т.п.). Контроллер служит «клеем» между моделью и представлением, передавая информацию между ними.

На основе идеи MVC, по моему мнению, была разработана диаграмма пригодности – полезному дополнению к языку UML (но, к сожалению, не вошедшему в официальную спецификацию).

Расслоение системы и относительная изолированность уровней облегчает независимую разработку слоёв и сопровождение системы. Не менее полезной оказываются средства автоматической синхронизации данных между слоями (в .NET это называется binding).

7.3.10. ООП

Преимущества и недостатки ООП:

+: повторное использование кода (наследование, компоненты, библиотеки)

+: инкапсуляция информации

+: естественное моделирование объектов предметной области

+: устойчивость к изменениям требований

+: понятность кода (он читается «как будто» на естественном языке)

-: снижение скорости выполнения методов

-: увеличение размера программы

-: повышение сложности системы

-: значительный рост требований к квалификации разработчиков

ООП давно зарекомендовала как продвинутая методика разработки ПО. В первую очередь из-за эффективного разделения понятий предметной области и системных представлений. Возможно, что за технологиями программирования, еще более усиливающих этот разрыв (ФП – функциональное, АОП – аспектно-ориентированное, ЛП – логическое) будущее.

Но, как сказано с чисто американским прагматизмом в статье [7], «доказывать свою правоту нужно делом, особенно когда тебя не хотят слышать». Многие слышали про перспективы логических языков в разработке ПК 5-го поколения в Японии. И где можно увидеть *реализации* этих ПК? Бумажный тигр не чета настоящему. Поэтому, на сегодняшний момент, ООП наиболее жизнеспособная технология создания ПО.

7.4. Этап ЖЦ «Кодирование»

7.4.1. Стандарт кодирования

Стандарт кодирования представляет собой документ, в котором описаны соглашения о структуре и организации программного кода.

Наиболее важные его элементы:

- наименования программных объектов (компонентов, модулей, классов, переменных, констант, методов)
- используемые префиксы (cb – **C**ombo**B**ox, tb – **T**ext**B**ox, btn – **B**u**t**to**n**, венгерская нотация)
- регистры символов - **c**amel**S**tyle, **P**ascal**S**tyle, **c**_plus_**p**lus_**s**tyle, ANOTHER_CPP_STYLE
- оформление и структура программы (отступы, табуляция, объявления)
- формат комментариев (шаблон спецификации классов, методов, объяснения алгоритмов)
- обработка исключений
- работа с памятью
- оформление структур типа if/then/else, switch, for/foreach/do/while … и многое другое.

Стандарт кодирования достаточно тяжело внедрить (каждый программист привык писать в своём стиле), но после принятия этого соглашения, уже невозможно представить себе работу без него.

Главные преимущества стандарта:

- прививание «хорошего» стиля написания кода (легко написать код, понятный компилятору, значительно труднее – понятный человеку)
- уменьшение затрат на сопровождение ПО (весь код написан в одном стиле)
- дисциплинирование процесса разработки (и его участников – если сказано описать каждый параметр метода, то нужно делать это в 100% случаев, а не время от времени)

7.4.2. Совместное владение кодом

Это УПР (а также, родственная идея о поддержке версионности исходных текстов), по моему мнению, является одним из «столпов» методики XP. Заключается оно в том, что любой человек может поменять любую часть системы (в XP правда сказано о любой *паре* разработчиков).

На первый взгляд, это УПР – путь к анархии и развалу программного проекта. Но это не так.

Во-первых, совместное владение кодом трудно себе представить без соответствующих инструментов. На данный момент существует множество как коммерческих (VSS, ClearCase), так и бесплатных (StarTeam, CVS) продуктов поддержки версионности и совместной работы.

Во-вторых, общее владение кодом повышает ответственность каждого участника процесса. Если для общего результата нужно поменять что-либо (а разработчик обладает требуемой квалификацией), то это нужно делать без длительных согласований.

Инструменты версионности поддерживают такие операции над файлами, как Check-Out (взять версию из хранилища на доработку), Check-In (положить измененную локальную версию в хранилище для общего доступа) и самое главное Merge (слияние *локальной* версии с *общей* версией при Check-In). Если система не может провести слияние в автоматическом режиме, она предлагает сделать это разработчику, выполняющего Check-In.

Системы версионности обычно интегрируются в среду разработки (главная функция – «Get latest version» - получение последней версии из хранилища). Следует также отметить, что системы обычно работают с *текстовыми* форматами файлов.

7.4.3. Пилот-проект

Разработка прототипа пользовательского интерфейса (пилот-проекта) *до начала* кодирования экономит большое количество времени разработчика и конечных пользователей.

Существуют автоматизированные инструменты для построения таких макетов (например, MS Visio), но, на мой взгляд, пока ничего лучше, чем карандаш, резинка и лист бумаги не придумано.

Когда пользователь видит нарисованные «на скорую руку» будущие формы приложения и их взаимодействие (в виде стрелок), он не боится что-либо менять в документе. Если же представить юзеру красиво оформленные снимки экранов («выглядящие как настоящие») или наполовину функционирующие приложение (с заглушками вместо вызова методов), то может создаться ложное впечатление о почти полной готовности системы.

Главная цель составление прототипа – получение быстрого ответа на вопрос: отвечает ли понимание разработчиков целям бизнес-требований и поиск путей для более быстрого получения юзером полезного эффекта от системы.

7.4.4. Острый инструмент

Недавно мне рассказали про молоток, который значительно облегчает труд строителя. Внутри он полый, а пустое пространство наполовину содержит твердые частицы. В момент удара, на несколько мгновений позже, «прилетают» частицы и гасят энергию отскока. Таким образом, повышается удобство в использовании инструмента.

То же самое в программировании. Необходимо найти и внедрить самую удобную и быструю среду разработки, компилятор, библиотеки функций и наборы компонент.

Не менее важным оказывается чисто физиологическое удобство работы со средой – цветовая гамма редактора кода, расположение панелей инструментов, «горячие» клавиши доступа к наиболее используемым функциям.

Надо признать, что многие IDE (интегрированные среды разработки) постоянно повышают эти нефункциональные (но очень важные) возможности. Сейчас уже невозможно представить себе среду без технологии IntelliSense (подстановка названия по первым набранным буквам), закладок, подсветки различных элементов программы, формирования шаблонов кода и развитого текстового редактора.

7.4.5. Структура данных

Хорошее знание SQL и способность понять код запроса, занимающий целый лист формата A4, несомненно украшает каждого разработчика, но зачем такие трудности? Если к данным приходится добираться, используя огромное количество Join-ов, то, возможно, выбрана неоптимальная форма хранения информации. Поэтому, иногда лучше изменить структуру данных (нормализовать или наоборот, денормализовать базу данных).

Вообще это относится не только к SQL. Во время разработки отчета в CrystalReports обнаружилось, что снизу к бланку заказа нужно ещё добавить корешок бланка. Я уже стал подумывать о субрепорте, его связи с главным отчетом и т.п., т.е. задача решалась непросто.

Но мне было подсказано поистине гениальное решение (его можно объяснить только большим опытом работы или богатым воображением). Решение заключалось в том, что нужно «перевернуть» отчет на бок и реализовать две части отчета в формате Landscape. После этого, две части одинакового Details нужно просто продублировать.

7.4.6. Тестовые проекты

В разработке ПО стоит полагаться на жизнеспособные, испытанные решения, не раз опробованные на практике. В условиях постоянной нехватки времени, которыми характеризуются практически все проекты, возрастает цена ошибки за неверно выбранную платформу, среду разработки, СУБД, технологию, язык программирования и т.п.

Поэтому, я считаю очень важным перед тем, как начать использовать новую технологию (какой бы революционной она не казалось) выделить ресурсы для хотя бы поверхностного изучения предмета. Наиболее хорошо для этого подходят тестовые проекты.

Их цель - в ограниченное время получить информацию о достоинствах технологии (скорость, легкость использования и разработки, ресурсоемкость, производительность, устойчивость, масштабируемость), а также о её недостатках и способах преодоления этих ограничений.

После проведения исследований, и получения положительных отзывов, участники эксперимента станут наставниками основной массы разработчиков. Такие опытные работы могут проводиться в промежутках между проектами и быть своеобразной формой поощрения за хорошую работу. Каждому хочется кроме рутинной работы, познакомиться с новыми веяниями в мире ИТ.

7.4.7. Парное программирование

ПП заключается в параллельной работе двух разработчиков над одним участком системы (см. статью [14]). На первый взгляд, это напрасная трата ресурсов (в первую очередь, времени). К тому же, многие представители профессий в ИТ просто не умеют работать вместе.

Я не считаю «чистое» ПП такой уж хорошей идеей. В разработке системы всегда есть «рутинные» операции – написание SQL-кода, разработка пользовательского интерфейса, методичное тестирование, отладка. Для таких видов работ лучше сконцентрироваться и сделать их самостоятельно.

Наоборот, решения по архитектуре системы (иерархия и взаимодействие классов, наследование, дизайн БД, модификация модели из-за изменений требований) лучше принимать сообща. Причем, лучше обсуждать их не вдвоем, а большим количеством человек (до 5).

Перечислю выгоды ПП:

- ошибки обнаруживаются на более ранних этапах, когда их легче и дешевле исправить
- уменьшается общее количество ошибок (в среднем на 15%, кстати, время разработки также увеличивается на 15%)
- улучшение дизайна системы, более оптимальные решения и меньшее количество кода из-за мозгового штурма
- трудности быстрее разрешаются
- увеличение обучаемости сотрудников
- повышение объема знаний о системе каждым разработчиком
- больше удовлетворения от работы

7.4.8. Рефакторинг кода

В статье [13] упоминается о рефакторинге (Р), как о самой мощной концепции, появившейся за последние 2000 лет. Я думаю, что это преувеличение, хотя нельзя не признать огромные выгоды от самого процесса осознания программистами этой идеи.

Р представляет собой перестройку кода без внешнего изменения функциональности (см. книгу [14]). Да, именно так просто. Это то, чем занимался каждый программист, в тот момент, когда начинал испытывать дискомфорт из-за ранее написанного неопрятного кода (код «с душком»).

Часто программисты пишут не идеальный, а *достаточно хороший код* и только позже, когда видят дублирование кода, запутанную логику, длинные методы начинают наводить порядок.

Основные элементы Р, это собственно каталог рефакторингов (согласно правилу Парето, 20% из них дают 80% эффекта), и принципы:

- вносить изменения небольшими порциями
- тестирование до Р и после
- если нужно внести новую функциональность, но это сложно (требуется Р), то лучше сначала сделать Р, а потом писать новый код
- правило «трех ударов» - сначала нужно создавать любой код, удовлетворяющий тестам; затем; встретившись с тем же кодом и поняв его недостатки, надо призадуматься; после третьего раза нужно начинать Р.

Также, следует отметить появление автоматизированных средств Р (Р-браузер).

7.4.9. Инкрементная разработка

Сейчас, казалось бы, уже никто не использует модель «водопада». Но часто это только декларируемое понятие, а не реальность.

В большинстве организаций (особенно крупных) существует четкая функциональная декомпозиция, т.е. каждый отдел выполняет узкий участок работы. Поэтому, *кажется*, что намного проще и эффективнее вести работу, разбив её на участки. И довести проект до успешного завершения, поочередно выполнив каждый этап: анализ, проектирование, построение БД, кодирование, тестирование и внедрение.

Такая модель основывается на предположении, что стоимость исправления ошибки экспоненциально возрастает по мере роста номера этапа. Т.е. исправить ошибку, выявленную на этапе тестирования (или еще хуже сопровождения) стоит дороже, чем во время кодирования. Поэтому целесообразно провести максимально качественно каждый этап, чтобы «не пропустить» ошибки дальше («большой дизайн» в начале разработки).

Однако жизнь вносит коррективы в любые планы. Технологии, платформы и бизнес меняются так быстро, что время и ресурсы, потраченные на детальную спецификацию требований, например, посредством технического задания, кажутся мне совершенно бесполезными.

Конечно, без предварительно проектирования *каркаса системы* вряд ли получится хорошая архитектура. Но решение проблемы постоянного изменения бизнес-требований сейчас я вижу только в постепенном *наращивании* продукта, в противовес его *строительству*.

7.5. Этап ЖЦ «Тестирование»

7.5.1. Постоянное тестирование

Возможно, на меня сильно повлияло участие в последних проектах «в стиле XP», но теперь я значительно терпимее отношусь к «смешиванию понятий» (этапов ЖЦ) в процессе разработки проекта. Я считаю весьма эффективным подвергать немедленному тестированию каждый разработанный элемент, перед непосредственной интеграцией в общую систему.

Например, после разработки нового DALC-а (Data Access Logic Component-а), уместно написать тест (обычного консольного приложения будет достаточно), в котором будут проверена инициализация компонента, работа с базой данных и методы бизнес-логики. В отладчике (или любым другим способом) необходимо проверять соответствие внутреннего состояния объекта ожидаемым результатам.

Такое «упрощенное» тестирование позволяет отвлечься от сложного графического интерфейса и обеспечивает возможность проверить «саму суть» нового элемента объектной модели.

Особенную выгоду этот метод дает, когда необходимо быть уверенным в корректном функционировании элемента системы использующегося с большим количеством начальных условий (например, алгоритмы).

Для автоматизации такого тестирования были разработаны инструменты (в их названии есть слово «Unit» - JUnit, NUnit, и т.д. Префикс характеризует язык/среду разработки тестируемой программы). Посредством рефлексии приложение распознает и запускает тестовые методы классов.

7.5.2. Автоматизация тестов

Хотелось бы надеяться, что времена, когда работа тестировщика представляла собой ежедневный монотонный ввод одних и тех же данных, с целью получения реакции системы, давно прошли. Сейчас, тестировщик вооружен не хуже, чем программисты, а зачастую и лучше их. На сегодняшний момент существует большой выбор инструментов для тестирования:

- Обнаружение утечек памяти
- Определение охвата тестируемого кода
- Выявление скорости работы методов
- Тестирование пользовательского интерфейса
- Нагрузочное тестирование

Большим преимуществом в работе отдела QA является наличие средств, позволяющих автоматизировать процесс тестирования. После добавления новых функций в систему очень важно убедиться, что не нарушена работа существующих функций, и для этого подходит система, самостоятельно «прогоняющая» большинство контрольных примеров без участия человека (например, в нерабочее время).

Независимое тестирование (программой, а не человеком), совместно с принципом «кто ломает сборку, тот находит и исправляет баг», по моему мнению, увеличивает персональную ответственность разработчика при интеграции своего кода в общую систему.

7.5.3. «Узкие» тесты

При тестировании системы, целесообразно добавлять компоненты системы по одному, чтобы сфокусироваться на «узком участке». Таким образом, детектирование ошибок одного модуля не влияет на результаты другого. Ошибки не будут испытывать взаимовлияние друг друга и их проще будет обнаруживать.

Это похоже на инкрементную разработку в той части, что всегда проще решить конечную задачу, разделив её на мелкие части.

Вообще, если при малейшем изменении начинается шквал ошибок, это признак того, что «Там не просто живут баги. Там их гнездо» (см. статью [33]). Такая ситуация сигнализирует о больших проблемах разработанного ПО.

7.5.4. Набор данных

Для проведения качественного процесса тестирования важно обладать полным набором данных, представляющих различные варианты ситуации автоматизируемого процесса. Чаще всего, такой набор данных предоставляет конечный пользователь системы, но во многих случаях, разработчики сами могут дополнить предоставленный набор своими вариантами исключительных ситуаций.

Например, пользователю не придет в голову проверить реакцию системы на отключение электричества, обрыв связи или внезапную недоступность СУБД посередине бизнес-транзакции.

Следует отметить, что тестирование нужно проводить на «эталонных» наборах данных, а не на данных разработчиков. Т.е. бизнес-сущности должны «по всем правилам» быть занесены в систему через соответствующую функциональность, а не сгенерированы «магической кнопкой».

Часто сгенерированные данные или занесенные «вручную» в БД, это *не те же самые данные*, которые позволяет создать система «официальным» способом. В отношения *качества данных*, различия, в первую очередь, проявляются в значениях по умолчанию отдельных полей. Иногда есть различия в том, что позволяет ввести СУБД и программа.

7.5.5. Окружение программы

Широко известно выражение типичного разработчика, в качестве «универсального ответа» на сообщение об ошибке на компьютере клиента. «На моей машине всё работало … по крайней мере 5 минут назад. Проблема у *них*, а не у *меня*». Проблема на самом деле у тестировщиков, в должной мере не просчитавших все (или почти все) ситуации окружения (environment) в котором была проинсталлирована программа.

Под окружением понимается та среда, в которой работает приложение. Это, во-первых, аппаратное обеспечение – компьютер (разной степени мощности), монитор (разной диагонали и разрешения), периферийные устройства, комплектующие и вообще всё, что имеет отношение к «железу». Во-вторых, это программное обеспечение – операционная система и патчи (сервис-паки к ней), драйвера, JVM, офисные пакеты, СУБД и доступ к данным, средства получения отчетности, коммуникации и обслуживания и т.п.

Необходимо обеспечить всестороннее тестирование системы во всевозможных программных и аппаратных конфигурациях. Если же программа предназначена для работы со строго определённой версией того или иного компонента, нужно максимально облегчить доступ пользователя к нему. Например, если приложение для работы требует установленного компонента X версии Y, то лучше включить его в инсталляционный пакет, чем ожидать, что пользователь установит его корректно самостоятельно.

7.5.6. Отслеживание ошибок

Системы по defect-tracking-у получили широкое распространение в последнее время и являются замечательным примером того, как реализация простой идеи может принести большую пользу. Суть таких инструментов состоит в том, чтобы увеличить контроль и довести до автоматизма процесс обнаружения, исправления и тестирования ошибок в ПО.

Разработчик, обнаруживший ошибку (владелец), даёт ей кодовое имя, описывает её симптомы, и ситуацию, приведшую к возникновению ошибки. Менеджер проекта назначает приоритет, сроки и ответственного за исправление проблемы. Назначенный работник, решает проблему и сообщает об этом «владельцу» ошибки. Тот удостоверяется, что вопрос решен и «закрывает» задачу.

Всё это можно делать в «бумажном» виде, например, в документе MS Word, но системы, интегрирующие в себе весь процесс накопления данных об ошибках, уведомлениях участников процесса (по email), получения отчетности, значительно увеличивают эффективность работы.

Последующий анализ ошибок, функции их распределения по времени, сортировка с учетом приоритета, критичности, частоты повторения, среднему времени исправления, помогут в дальнейшем если не предотвратить их появление, то хотя бы понять причины их возникновения (и принять соответствующие меры).

7.5.7. Юзабилити

Зачастую, под тестированием ПО, понимается только тестирование функциональности, производительности, масштабируемости, устойчивости к сбоям и прочие «серьёзные» вещи. При этом забывают об эстетическом восприятии программы пользователем и удобстве его работы.

Конечно, для процесса оформления новой продажи, главное – это сама возможность выбрать товар из списка доступных, напечатать чек и оплатить покупку способом, удобным покупателю. Но если «основной» функциональный эффект достигнут, то стоит потратить некоторые ресурсы, чтобы сделать процесс удобным. Например, посчитать и выделить сумму покупки, при вводе нового товара сделать автоподстановку названия по первым введенным буквам (а не заставлять выбирать товар из всего ассортимента), обеспечить поддержку «горячих» клавиш и т.п.

В книге [11] описаны сравнительно простые, но эффективные способы повышения субъективного удовлетворения от работы с программой.

Может оказаться полезным снять на видео работу пользователей с приложением, для определения того, на какие виды работы более всего тратиться их время.

Хорошим примером того, что «программы нужно делать для юзеров, а не для программистов», является популярные сейчас способы работы с приложениями с помощью «визардов». В этом есть большая заслуга компании Microsoft. Чтобы научиться использовать её продукты, даже такие сложные как СУБД, тратиться на порядок меньше времени, чем с продукцией конкурентов.

8. Заключение

Ну, вот в принципе и всё. Быстро книга закончилась, правда? Я тоже люблю короткие книги. Мне всегда нравились «ламерские» книги, в отличие от «библий пользователя» технологии X. Зачем читать толстенные талмуды, если можно потратить 5 минут, чтобы просмотреть Getting Started и начать пользоваться продуктом, а проблемы можно решить и в «рабочем порядке».

Конечно, можно высказать критику по поводу отсутствия в книге *полной картины* всего жизненного цикла программного продукта, от рождения и до смерти. Но у меня действительно были запланированы ещё несколько глав книги.

В главе «ЖЦ Внедрение» я хотел рассказать о таких УПР, как методики «смягчения стресса» для юзеров в процессе перехода к новому продукту. Качественная документация, обучение, конвертирование данных из «старой» системы в «новую» должны облегчить процесс перехода.

Тесная связь с конечными пользователями, своевременное реагирование на обнаруженные проблемы посредством патчей и хотфиксов. Хранение и контроль версий успешных и тестовых билдов продукта, ведение истории внедрения для каждого клиента. Создание инсталляционного пакета и процесс обновления компонентов системы.

В главе «ЖЦ Сопровождение» я хотел рассказать о методике внесения изменений в существующее ПО. Т.к. сопровожденцы часто не первоначальные разработчики, то это весьма важный процесс. Если малозначительные изменения в интерфейсе, отчетах и формате данных система

еще может «стерпеть», то при накоплении критической массы (энтропии), иногда проще всё бросить и переписать заново.

Ощутимым конкурентным преимуществом является служба «горячей поддержки» клиентов. Пользователю ПО приятно будет знать, что в любое время дня и ночи о нём знают, помнят и готовы помочь. Конечно, с развитием бизнеса компании, каждый отдельный клиент вносит всё менее значимый вклад (в процентном отношении) в благосостояние компании, но это как лавина – уйдет один, за ним потянутся другие, а там и до банкротства недалеко.

Всё это могло бы быть написано, если бы не одно обстоятельство. Дело в том, что я чрезвычайно азартный и увлекающийся человек. Поэтому, начав разрабатывать тему УПР, я никак не смог остановиться (хотя, если вы читаете эту главу, значит всё-таки смог ;-).

После вышеперечисленных двух этапов ЖЦ ПО, я подумал об еще одном – рекламе продукта. Операционная система – это не что-то среднее между шампунем и памперсами. Вряд ли можно увеличить популярность ПО рекламой типа «В каждом седьмом экземпляре текстового процессора – неизвестный баг, найди его и выиграй поездку в Редмонд за собственные деньги!» ;-). «Сообщение о вызове в Америку появляется уже через полчаса работы с программой!» Реклама ПО – отдельная тема, которая еще ждёт своих исследователей.

После рекламы я подумал о процессе продажи ПО, стратегии развития и интеграции с другими продуктами и т.п.

В результате, я пришел к выводу, что эту книгу можно писать целую вечность, что не входит в мои планы (кроме книги у меня ещё есть работа с 9 до 18). И так написание чернового варианта заняло более трех месяцев (при запланированных двух). А каждый день выходит столько интересных книг, статей, версий продуктов… и всё это нужно прочесть, проанализировать, попробовать…

Поэтому, я решил сконцентрироваться только на этапах собственно *производства* продукта. Все изменения, которые мне придут в голову, полученные рецензии, а также результаты моего дальнейшего профессионального развития, скорее всего, войдут во второе издание книги (при наличии на то свободного времени).

Теперь заключительное слово. Когда большая часть материала была написана, ко мне попала статья [34]. Я ожидал в ней увидеть знакомые фамилии – Брукс, Коберн, Фаулер и т.п. Но я ошибся. Ни одна из этих книг не была мне знакома. Фамилия McConnell всё же рождала какие-то смутные ассоциации, но в основном связанные с маркой кофе.

Я понял, что чем больше я стараюсь узнать мир, тем более широкий горизонт мне открывается и знания оказываются совершенно безграничными. Когда-то изучив книгу по Turbo Pascal 7.0, я считал себя крутым гуру ;-). Теперь же я могу признать, что «чудесное затруднение» (см. книгу [6]) связанное с полной невозможностью осилить хоть какой-нибудь большой объем знаний в данной научной дисциплине, делают профессию программиста совершенно замечательной.

Я думаю, что большая часть УПР ещё неизвестна широкой публике. К счастью, есть люди, которые стараются сделать жизнь лучше. И это не *особенные* люди – у *каждого* должно быть достаточно сил, чтобы на несколько миллиметров поднять общую планку развития.

Не боги горшки обжигают. Поэтому я верю, что у каждого (в том числе и у меня) хватит сил, чтобы осуществить «повышение производительности, которое позволяет делиться избыточным, а не драться за недостающее» (см. эпиграф к книге [8]).

9. Библиография

Книги:

1. Дуг Розенберг, Кендалл Скотт, «Применение объектного моделирования с использованием UML и анализ прецедентов», М., ДМК, 2002
2. Дин Леффингуэл, Дон Уидриг, «Принципы работы с требованиями к программному обеспечению», М., Вильямс, 2002
3. Алистер Коберн, «Современные методы описания функциональных требований к системам», М., Лори, 2002
4. Алистер Коберн, «Быстрая разработка программного обеспечения», М., Лори, 2002
5. Эдвард Йордон, «Путь камикадзе», М., Лори, 2000
6. Фредерик Брукс, «Мифический человеко-месяц», Спб, Символ, 2001
7. Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес, «Приёмы объектно-ориентированного проектирования», Спб, Питер, 2001
8. Гради Буч, «Объектно-ориентированный анализ и проектирование», М., Бином, 2000
9. Крэг Ларман, «Применение UML и шаблонов проектирования», М., Вильямс, 2001
10. Эд Салливан, «Время - деньги», М., Русская редакция, 2002

11. Влад Головач, «Дизайн пользовательского интерфейса», компания Usethics, электронная книга
12. П. Коуд, Д. Норт, М. Мейфилд, «Объектные модели», М., Лори, 1999
13. Ален Голуб, «Верёвка достаточной длины, чтобы выстрелить себе в ногу», электронная книга
14. Мартин Фаулер «Рефакторинг», Спб, Символ, 2004
15. Ли Якокка «Карьера менеджера», электронная книга
16. Кент Бек, Мартин Фаулер «Экстремальное программирование: планирование», Спб, Питер, 2003
17. Джордж Влиссидес «Применение шаблонов проектирования», М., Вильямс, 2003
18. Т. Бадд «Объектно-ориентированное программирование», электронная книга
19. Мартин Фаулер «Архитектура корпоративных программных приложений», М., Вильямс, 2004
20. Р. Тротт, А. Шаллоуэй, «Шаблоны Проектирования»
21. Дж. Бентли, «Жемчужины программирования, Спб, Питер, 2002
22. Т. Кормен, Ч. Лейзерсон, Р. Ривест, «Алгоритмы. Построение и анализ», М., МЦНМО, 2001
23. Дональд Кнут, «Искусство программирования для ЭВМ», (Т1 – Основные алгоритмы; Т2 – Получисленные

алгоритмы; Т3 – Сортировка и поиск), М., Мир, 1976(7)(8).
24. Дэвид А. Марка, Клемент МакГоуэн, «Методология структурного анализа и проектирования SADT», электронная книга

Сайты:

1. www.interface.ru – документация, отличный учебный центр
2. www.gotdotnet.ru – все о среде .NET
3. www.microsoft.com – продукты Microsoft. Можно как угодно плохо относится к этой фирме (и к её лидеру), но заслуга компании по стандартизации ПО и развитию всей отрасли, представляется мне совершенно безграничной
4. www.xprogramming.ru – наиболее известный российский сайт методики экстремального программирования XP
5. www.maxkir.com – отличные переводы «культовых статей», ну и XP конечно
6. www.osp.ru – один из самых лучших онлайновых ИТ-журналов в России
7. www.citforum.ru – море всевозможной документации. Неоценимые заслуги в деле поиска информации для рефератов, статей, книг и т.п.
8. www.optim.ru – неплохой российский онлайн ИТ-журнал
9. www.rational.com – без комментариев. Всё о продуктах Rational, методике RUP и языку UML

10. www.therationaledge.com – журнал, освещающий последние события «в мире Rational»
11. www.rsdn.ru – отличная база знаний о программировании и не только
12. www.dotsite.spb.ru – все об языке C# и среде .NET
13. www.sql.ru – самый известный ресурс, связанный с СУБД. Можно потратить целую жизнь на прочтение веток форумов
14. http://russian.joelonsoftware.com/ - тонкие, юмористические и философские замечания о создании ПО (хотя и не без похвалы в адрес *своей* компании) Джоэла Сполски
15. www.xcomx.narod.ru – все об XCOM, пришельцах и Джиллиан Андерсон.
16. www.softcraft.ru – разноликое программирование. Заставляет поверить в преимущества российских программистов перед индийцами, китайцами и прочими «оффшорными гигантами»
17. www.caseclub.ru – все о CASE
18. www.tester.com.ua – тестирование
19. www.refactoring.com – рефакторинг как он есть
20. www.omg.org – стандарты, спецификации
21. www.1001.vdv.ru (раздел «Соло на клавиатуре») – если бы не метод слепого набора текста, эта книга появилась бы на свет несколькими месяцами позже (а возможно, вообще не появилось, т.к. моё терпение закончилось бы раньше)

22. www.cetus-links.com – всевозможные ссылки на полезные сайты «мира ИТ»
23. http://www.microsoft.com/rus/msdn/msf/ - методология Microsoft Solutions Framework
24. http://is.twi.tudelft.nl/~hommes/toolsub.html - самая широкая классификация методик для бизнес-моделирования и продуктов, поддерживающих их

Статьи:

1. http://www.osp.ru/os/2002/10/039.htm – «Как добиться успеха в безнадежных проектах», Журнал "Открытые системы", #10, 2002г., Константин Берлинский
2. http://www.rsdn.ru/article/career/CareerCalculus.xml - «Формула успеха», Автор Eric Sink, Перевод: Лев Курц
3. www.softcraft.ru/design/moving.shtml – «Новая инициатива в программировании. Движение за открытую проектную документацию», 2003г., А.А. Шалыто
4. http://www.interface.ru/fset.asp?Url=/rational/014.htm, «Rational Rose, BPwin и другие — аспект анализа бизнес-процессов», Журнал "Директору информационной службы", #11/2000, Павел Сахаров
5. http://www.interface.ru/fset.asp?Url=/erp/news/m010628491.htm, «Автоматизация хаоса», Андрей Акопянц
6. http://www.gotdotnet.ru/LearnDotNet/NETFramework/592.aspx, «Проектирование компонентов уровня данных и передача данных между уровнями», Microsoft

Corporation, 2002г., Анджела Крокер (Angela Crocker), Энди Олсен (Andy Olsen) и Эдвард Джезирски (Edward Jezierski)
7. http://www.osp.ru/pcworld/2004/03/102_print.htm, «Никлаус Вирт — патриарх надежного программирования», Мир ПК, #03/2004, Руслан Богатырев
8. http://www.osp.ru/cio/2004/04/080_print.htm, «Взрывная волна CMM», Директор ИС, #04/2004, Кристофер Кох
9. http://www.osp.ru/cio/2004/04/088_print.htm, «Модель предметной области», Директор ИС, #04/2004, Дмитрий Цуцаев, Андрей Алексеенко
10. http://www.osp.ru/cw/2004/16/054_1_print.htm, «Цель — собрать команду проекта», Computerworld, #16/2004, Кэтлин Меламьюка
11. http://www.osp.ru/os/2004/02/072_print.htm, «Используют ли программисты документацию?», Открытые системы, #02/2004, Тимоти Летбридж, Джанис Сингер, Эндрю Форвард
12. http://xprogramming.com.ua/advantages.php, «Преимущества XP перед другими известными методологиями разработки», Александр Федоренко
13. http://xprogramming.ru/Articles/MisconceptionXP-II.html, «Ахиллес и Черепаха рассуждают о Дизайне», Chaos Engineering, Брайан Доллери
14. http://maxkir.com/sd/pairprog_RUS.htm, «Парное программирование:

преимущества и недостатки», Алистэр Коуберн
15. http://xprogramming.ru/Articles/CodeSmells.html, «Дурно пахнущий код», Руслан Ерёмин
16. http://maxkir.com/sd/explicit_RUS.html, «Чтобы было яснее», ThoughtWorks, Мартин Фаулер
17. http://maxkir.com/sd/newmethRUS.html, «Новые методологии программирования», ThoughtWorks, Мартин Фаулер
18. http://maxkir.com/sd/justintimemethodologyconstruction_RUS.html, «Каждой методологии - свое время», Humans and Technology Technical Report, 2000.01, Алистэр Коуберн
19. http://maxkir.com/sd/RLCD_highsmith.html, «Устаревшие методологии - на пенсию!», Software Testing & Quality Engineering, Июль/Август 2000, Джим Хайсмит
20. http://maxkir.com/sd/designDead_RUS.html, «Проектирования больше нет?», ThoughtWorks, Мартин Фаулер
21. http://maxkir.com/sd/useCasesTenYearsLater_RUS.html, «Варианты использования, десять лет спустя», Humans and Technology, Журнал STQE, Март/Апрель 2002 г., Алистэр Коуберн
22. http://maxkir.com/sd/methyperproject_RUS.htm, «Каждому проекту своя методология», Humans and Technology Technical Report 04.1999, Алистэр Коуберн

23. http://maxkir.com/sd/SilverBulletBitterPill_RUS.htm, «Серебряная пуля или горькая пилюля?», Кен Швабер
24. http://maxkir.com/sd/people_as_nonlinearRUS.htm, «Люди как нелинейные и наиболее важные компоненты в создании программного обеспечения», Humans and Technology, 10.1999, Алистэр Коуберн
25. http://maxkir.com/sd/expDocumentationInXP_RUS.html, «Основы Extreme Programming: документация», 11/21/2001, Рон Джеффриз
26. http://maxkir.com/sd/manualsInXp_RUS.html, «Extreme Programming и руководство пользователя», 11/21/2001, Рон Джеффриз
27. http://maxkir.com/sd/testing.html, «Организация и именование автоматизированных тестов», февраль-март, 2003, Кирилл Максимов
28. http://xprogramming.ru/Articles/ExtremeTesting.html, «Экстремальное тестирование», Роман Ерёмин
29. http://xprogramming.ru/Articles/CustomerInXP.html, «Заказчик. Повадки и особенности», Роман Ерёмин
30. http://xprogramming.ru/Articles/XPEpisode.html, «Эпизод Экстремального Программирования», Роберт Мартин и Роберт Косс
31. http://xprogramming.ru/Articles/LoveUT.html, «Учимся любить юнит тесты», Перевод статьи из журнала STQE, PDF версия - на

сайте Pragmatic Programmer, Дейв Томас и Энди Хант
32. http://xprogramming.ru/Articles/TDD-PERL.html, «Основанная на Тестировании Разработка. Пример на языке PERL», Денис Косых
33. http://nosorog.org/cgi-bin/statyipk.pl?nm=1069802969, «Программисты и Космополитизм», автор неизвестен
34. http://www.silicontaiga.ru/home.asp?artId=2429, «Десять программистских книг, которые потрясли мир, но все еще неизвестны в России», Компьютерра, Андрей Терехов
35. http://www.klerk.ru/soft/1c/?1880, «Системы управления знаниями и непрерывное управление жизненным циклом корпоративной информационной системы», 15.01.2003, Кабанов Алексей
36. http://www.nsda.ru/home.asp?artId=2142 , «Требования к информационной системе и модели жизненного цикла», Carabi Solutions, 11/12/2003, Колтунова Екатерина
37. http://www.interface.ru/rational/teh.htm, «Технология разработки программного обеспечения», сайт "Корпоративные системы", 1/2002, Дмитрий Безуглый
38. http://www.vernikov.ru/material89.html, «Какой CASE - инструмент нанесет наименьший вред организации?», 2001, Сергей Рубцов

10. Авторские права

В отношении данного материала действуют следующие правила:

1. только автор может вносить изменения в текст книги;
2. публикация в электронной версии не требует согласования автора;
3. перевод текста и/или публикация в письменном виде требует разрешения владельца авторских прав.

По поводу этой книги с автором можно связаться по адресу: **nsp_book@mail.ru**

Принимается любая конструктивная критика, помощь в оформлении, литературной коррекции текста и предложения о сотрудничестве.

С уважением,
Берлинский Константин,
разработчик ПО ИТ-отдела
компании Maximum,
Республика Молдова