ПРОФЕССИОНАЛЬНО

Роберт Гласс

ФАКТЫ

И ЗАБЛУЖДЕНИЯ ПРОФЕССИОНАЛЬНОГО ПРОГРАММИРОВАНИЯ

СИМВОЛ®

Facts and Fallacies of Software Engineering

Robert L. Glass

Addison-Wesley

ПРОФЕССИОНАЛЬНО

Факты и заблуждения профессионального программирования

Роберт Гласс

СИМВОЛ®

Санкт-Петербург — Москва
2008

Серия «Профессионально»

Роберт Гласс

Факты и заблуждения профессионального программирования

Перевод В. Овчинникова

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Выпускающий редактор	*В. Овчинников*
Научный редактор	*М. Деркачев*
Художник	*В. Гренда*
Корректор	*И. Губченко*
Верстка	*О. Макарова*

Гласс Р.
Факты и заблуждения профессионального программирования. – Пер. с англ. – СПб.: Символ-Плюс, 2007. – 240 с., ил.
ISBN 13: 978-5-93286-092-2
ISBN 10: 5-93286-092-8

Автор, имеющий огромный опыт работы в индустрии ПО, посвятил свой труд ее фактам, мифам и недоразумениям, представив 55 фактов и 10 заблуждений, относящихся к менеджменту, жизненному циклу, качеству, исследованиям и образованию в сфере разработки ПО. Некоторые из них хорошо известны, о других, наоборот, знают немногие. Основное внимание уделяется менеджменту как главной проблеме современной индустрии ПО, отрицательной роли рекламных кампаний, которые побуждают людей гоняться за миражами, и человеческому фактору – специалистам, без которых создание программ немыслимо.

Адресована широкому кругу читателей – от тех, кто управляет программными проектами, до программистов.

ISBN 10: 5-93286-092-8
ISBN 13: 978-5-93286-092-2
ISBN 0-321-11742-5 (англ)

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, edit@symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.
Подписано в печать 09.11.2007. Формат 70x90$^{1}/_{16}$. Печать офсетная.
Объем 15 печ. л. Тираж 3000 экз. Заказ N
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Эта книга посвящена тем исследователям,
которые зажгли огонь технологии программирования,
и тем практикам, которые поддерживают его горение.

Оглавление

Об авторе

Роберт Л. Гласс провел в вычислительных залах уже более 45 лет, а начал он с короткого трехлетнего периода работы в авиакосмической промышленности (в North American Aviation Inc.) с 1954 по 1957 г., что дает ему право называться одним из настоящих пионеров индустрии ПО.

После Северо-Американской он работал еще в нескольких авиакосмических компаниях (Aerojet-General Corp. в 1957–1965 гг. и Boeing Co. в 1965–1970 и 1972–1982 гг.). По большей части его работа заключалась в создании программных инструментальных средств, с которыми работали прикладные специалисты. Участвовать в авиакосмическом бизнесе в то время было делом волнующим – ведь это была эйфорически упоительная эпоха исследования космоса. Но работа в области вычислительной техники и программирования кружила головы еще больше. В обеих областях прогресс был стремительным, а перспективы неземными!

Главный урок, усвоенный им за годы, проведенные в авиакосмической отрасли, состоял в том, что ему очень нравилась техническая сторона индустрии ПО, но быть менеджером он совсем не хотел. Он старательно вживался в роль технического специалиста, и это сильно повлияло на его карьеру двояким образом:

1. его технические знания оставались свежими и пригодными к использованию, но
2. его компетентность как менеджера – и его возможности в смысле зарабатывания денег (!) – соответственно уменьшились.

Когда его способность продвигаться вверх по карьерной лестнице достигла неизбежного предела, он предпринял фланговый маневр, перейдя на научную и преподавательскую работу. Он читал курс лекций по инженерии ПО аспирантам Университета Сиэттла (1982–1987) и один год (1987–1988) проработал в (очень академическом) Институте инженерии

ПО (Software Engineering Institute – SEI). (До этого, получив грант, он два года (1970–1972) занимался исследованиями инструментальных средств в Вашингтонском университете.)

За эти годы научной и преподавательской работы Гласс извлек еще один главный урок. Его разум с восторгом обратился к научной стороне разработки программного обеспечения, но сердце так и осталось сердцем практика. Конечно, можно оторвать человека от его призвания, но нельзя вырвать призвание из его души. Вооружившись этой новой мудростью, он начал искать способ соединить академическую и практическую области вычислительной техники, перебросив мост через то, что он давно ощущал как «информационную пропасть».

И он нашел несколько способов. Многие из его книг (более 20) и статей (более 75) посвящены тому, как оценить открытия в вычислительной технике, сделанные учеными, и как внедрить в индустрию ПО те из них, которые имеют практическую ценность. (Это задача, бесспорно, нетривиальная, и именно она в значительной мере определяет уникальную и противоречивую природу его воззрений и печатных работ.) Читая лекции и проводя семинары, он сосредотачивается как на теоретических, так и на лучших практических достижениях, помогающих в реальной работе. Этому же посвящен и его бюллетень «The Software Practitioner», как и несколько более академический журнал «Journal of Systems and Software», который он редактировал много лет для издательства Elsevier (сейчас он его почетный редактор). И колонки, которые он ведет в таких изданиях, как «Communications of the ACM, IEEE Software» и ACM SIGMIS's «DATA BASE». Большинство его работ серьезны и самобытны, но изрядная их доля написана частично (а некоторые и абсолютно) в юмористическом ключе.

Какие же моменты его карьеры, если иметь все это в виду, можно назвать самыми торжественными? В 1995 г шведский университет Линкопинга присвоил ему почетную степень Ph.D, а в 1999 г. он был избран членом профессиональной ассоциации вычислительной техники ACM (Association for Computing Machinery).

Благодарности

Полу Бекеру (Paul Becker),

теперь работающему в Addison-Wesley, который редактировал почти все мои книги из числа опубликованных не самостоятельно, за то, что он верил в меня все эти годы.

Карлу Вигерсу (Karl Wiegers),

за его вклад в виде фундаментальных фактов, которые так часто забывают, и за его большую работу по рецензированию и наведению блеска на то, что я написал.

Джеймсу Баху (James Bach), Вику Бэсили (Vic Basili), Дейву Карду (Dave Card), Элу Дэвису (Al Davis), Тому Демарко (Tom DeMarco), Якову Фенстеру (Yaacov Fenster), Шери Лоуренсу Пфлигеру (Shari Lawrence Pfleeger), Денису Тейлору (Dennis Taylor) и Скотту Вудфилду (Scott Woodfield) за неоценимую помощь в поиске подходящих ссылок на источники изложенных фактов.

Предисловие

Когда я узнал, что Боб Гласс собирается написать эту книгу и сделать это по образцу моей «201 Principles of Software Development»,[1] то слегка призадумался. Боб один из лучших авторов в нашей отрасли, и его книга способна составить сильную конкуренцию моей. А когда Боб попросил меня написать введение, я заволновался – ведь надо одобрить книгу, которая вроде бы напрямую конкурирует с одной из моих? Однако после прочтения «Facts and Fallacies of Software Engineering» я рад и польщен предоставленной мне возможностью написать это вводное слово (и уже больше не волнуюсь!).

Индустрия ПО переживает сейчас то же, что переживала фармацевтика в конце XIX века. Порой кажется, что среди нас стало намного больше шарлатанов, продающих всяческие зелья и предсказывающих судьбу, чем нормальных людей, которые занимаются настоящим делом и несут знание в массы. Чуть ли не ежедневно мы слышим, что найдено замечательное новое решение непреодолимой проблемы. Мы много раз слышали о быстрых средствах от невысокой эффективности, низкого качества, недовольных клиентов, скверной связи, изменяющихся требований, неумелого тестирования, слабого руководства и т. д. и т. п. Подобных всезнаек развелось столько, что мы задаемся законным вопросом, а есть ли хоть доля правды в рассказах обо всех этих панацеях. Кого мы можем спросить? Кому мы можем доверять в нашей отрасли? Где узнать правду? Ответ: у Боба Гласса.

Боб уже много лет снабжает нас материалами для размышлений о многочисленных катастрофах, связанных с ПО. И я надеялся, что он соберет общие черты этих событий в один портрет, который нам было бы легче узнать, опираясь на богатый опыт автора. Пятьдесят пять фактов, которые

[1] Davis, A. M., «201 Principles of Software Development», McGraw-Hill, 1995.

здесь обсуждает Гласс, – это вовсе не его догадки., а как раз то, чего я ждал: мудрость, приобретенная автором при глубоком изучении сотен случаев, о которых он писал в прошлом.

Надо полагать, не всем читателям понравятся 55 фактов, приведенные в части I. Некоторые из них вступают в прямое противоречие с так называемыми общепринятыми «новыми веяниями». Тем из вас, кто предпочтет проигнорировать советы, содержащиеся на страницах этой книги, я могу лишь пожелать счастливого пути, но я опасаюсь за вашу безопасность. Вы далеко не первый, кто ступает по этой территории, густо усеянной минами, и многие сломали себе карьеру, пытаясь ее пройти. Лучший совет, который я могу вам дать – это прочесть любую из более ранних книг Боба Гласса, посвященных катастрофам, связанным с ПО. Те из вас, кто последует советам автора, тоже пойдут по хорошо проторенной дороге. Однако на этом пути вам встретится масса успешных примеров. Это путь знания и компетентности. Верьте Бобу Глассу, ибо он ступал по нему прежде. У него была и есть привилегия анализа своих успехов и неудач вместе с сотнями чужих успехов и поражений. Возьмите его опыт, и вы с большой вероятностью преуспеете в этой отрасли. Пренебрегите его советом и не удивляйтесь, если через несколько лет Боб попросит вас рассказать о вашем проекте, чтобы добавить его в свой следующий сборник рассказов о крушениях в разработке ПО.

Алан М. Дэвис
Весна 2002 г.

Дополнение автора:

Я пытался заставить Эла слегка снизить тон своего вступления. Все-таки оно получилось немного приторным. Но все мои попытки завершились крахом. (Я *действительно* пытался! Честное слово!) Отражая одну из них, он сказал: «Ты заслуживаешь быть на пьедестале, и я рад случаю помочь тебе подняться на него!» Мой опыт говорит, что за вознесением на пьедестал неизбежно следует падение, в результате которого ты разбиваешься, как Шалтай-Болтай, на мельчайшие обломки.

Все это правда, однако более замечательных и удивительных отзывов, чем те, какие здесь адресует мне Эл, я и представить не могу. Спасибо!

Роберт Л. Гласс
Лето 2002 г.

I

55 фактов

Введение

Эта книга представляет собой сборник фактов и заблуждений из области технологии программирования.

Звучит скучновато, не правда ли? Длинный список фактов о создании ПО не очень соблазняет на трату денег, а потом и времени. Но есть что-что особенное в этих фактах и заблуждениях. Они фундаментальны. И люди часто забывают истину, которая лежит в их основе. На самом деле она лежит в основе этой книги. Из того, что мы обязаны знать о создании ПО, мы многого по той или иной причине не знаем. А кое-что из того, что мы полагаем нам известным, попросту неверно.

Кто эти *мы* в предыдущем абзаце? Люди, которые создают ПО, конечно же. Мы, похоже, должны снова и снова учить все те же уроки – уроки, которых можно избежать, если, конечно, эти факты помнить. Но под *мы* я также понимаю тех, кто исследует программное обеспечение. Некоторые из них настолько глубоко уходят в теорию, что пропускают порой важные факты, способные перевернуть их теории с ног на голову.

Так что в аудиторию этой книги входит любой, кто интересуется созданием ПО. Профессионалы – как технические специалисты, так и их менеджеры. Студенты. Преподаватели. Исследователи. Я думаю без ложной скромности, что в этой книге найдется что-нибудь для всех из вас.

Первоначально книга имела громоздкое название из тринадцати слов: «Fifty-Five Frequently Forgotten Fundamental Facts (and a Few Fallacies) about Software Engineering» (Пятьдесят пять часто забываемых фундаментальных фактов (и несколько заблуждений) из области технологии программирования). Оно было, пожалуй, отмечено печатью неумеренности (по крайней мере, так рассудили те, кто отвечал за распространение книги). Так что здравый смысл одержал победу. Мы с моим издателем остановились в конце концов на варианте «Facts and Fallacies of Software Engineering» (Факты

и заблуждения из области технологии программирования). Четко, ясно, но совсем не так ярко!

Я пытался укоротить исходное длинное заглавие, превратив его в «The F-Book» (F-книга), чтобы обратить внимание на аллитерацию (повторение) буквы «F». Но издатель это предложение не одобрил, и я, наверное, должен признать его правоту. Буква «F», вероятно, единственная «грязная» буква[1] английского алфавита (некоторые выдвигают на эту позицию буквы «H» и «D», но «F», пожалуй, достигла в этом смысле качественно иного уровня). Так что это не «F-книга». (Одна из первых компьютерных книг, посвященная написанию компиляторов, называлась «Dragon Book»[2] по той единственной причине, что кому-то пришло в голову (это всего лишь моя догадка) поместить на обложку изображение дракона, но данное обстоятельство никак не помогло мне в споре).

В свою защиту хочу сказать, что каждое слово, начинающееся с буквы «F», имело свой смысл и каждое вносило свой вклад в суммарный смысл заглавия. Число 55 было, конечно, всего лишь уловкой, которая понадобилась мне, чтобы удлинить аллитерацию в названии. (Бьюсь об заклад, что число 201 в названии чудесной книги Алана Дэвиса о 201 принципе технологии программирования появилось настолько же случайно.) Но все остальные F-слова выбирались тщательно.

Часто забываемые (*frequently forgotten*). Потому что это справедливо для большинства из них. Здесь много фактов, о которых вы сможете сказать: «Ах да, помню-помню», а потом подумать о том, почему вы годами не вспоминали о них.

Фундаментальные (*fundamental*). Эти факты были выбраны главным образом потому, что они обладают особой важностью в программистской сфере. Пусть мы даже забыли многие из них, они от этого не стали менее значимыми. Если вы до сих пор не решили, стоит ли продолжать читать эту книгу, и хотите знать самую главную причину, по которой стоит это сделать, то скажу вам, что в этой коллекции фактов вы найдете (я в этом уверен) самые фундаментальные знания из области технологии программирования.

Факты (*facts*). Странно, но это, пожалуй, самое спорное слово в заглавии. Вы можете согласиться не со всеми фактами, которые я собрал здесь.

---

[1] F-word обозначает не только слово, начинающееся с буквы F, но и вообще слово, не рекомендуемое к употреблению в приличном обществе. – *Примеч. перев.*

[2] А. Ахо, Р. Сети, Дж. Д. Ульман «Компиляторы: принципы, технологии и инструменты», Вильямс, 2001 г.

Вы можете даже яростно протестовать против некоторых из них. Я лично полагаю, что все они соответствуют действительности, но это не означает, что вы должны думать так же.

Несколько заблуждений (*few fallacies*). Я не удержался от того, чтобы пронзить своей критикой несколько «священных коров» программирования! Думаю, я должен признать, что те вещи, которые я зову заблуждениями, другие могут назвать фактами. Но предполагается, что, читая эту книгу, вы должны получить удовольствие, и не последняя роль в этом отводится формированию вашего собственного мнения о том, что я называю фактами и заблуждениями.

А что можно сказать по поводу их актуальности? Рецензенсзируя книгу, один из моих коллег заметил, что некоторые из них устарели. С этим нельзя не согласиться Должно быть, часто забываемые сведения когда-то были хорошо известны. Старых хитов в этом сборнике предостаточно. Но некоторые факты и заблуждения могут удивить вас – так же, как незнакомые вам идеи (и оттого для вас новые). Так что дело не в том, что эти факты старые, а в том, что они нетленны.

Здесь я хочу представить вам те факты, о которых пойдет речь ниже. Часть II, посвященная заблуждениям, предваряется отдельным введением. Идея введения заключается в том, чтобы окинуть последний раз взглядом все эти 55 часто забываемых фундаментальных фактов и проследить, сколько из них связаны с ключевыми словами из первоначального заглавия книги. Рассуждая беспристрастно, я должен признать, что некоторые из этих фактов совсем не забыты.

- Двенадцать из них просто малоизвестны. Они не были забыты, многие о них даже не слышали. Однако они, по-моему, имеют фундаментальное значение.
- Одиннадцать из них достаточно хорошо распространены, но, похоже, никто не руководствуется ими в своих действиях.
- Восемь общепризнанны, но мы не приходим к единому мнению о том, как (и стоит ли вообще) решать проблемы, которые они олицетворяют.
- Шесть фактов, вероятно, не подвергаются сомнению большинством, и о них забывают редко.
- Пять фактов будут открыто оспариваться многими.
- Еще пять приняты многими, но некоторые ожесточенно опровергают их, что делает их достаточно спорными.

Это не дает в сумме 55, так как факты а) могут входить в несколько категорий и б) могут присутствовать в других категориях в ничтожных количествах (например, «только производители не согласились бы с этим»). Не буду классифицировать факты по категориям, а предоставлю вам возможность составить о них собственное мнение.

Нетрудно заметить, что данная книга изобилует спорными вопросами. Чтобы вам было легче в них разобраться, за каждым обсуждением факта я привожу сведения о полемике, которую он вызывает. Я надеюсь таким способом учесть и вашу точку зрения независимо от того, совпадает ли она с моей, а кроме того, вы увидите, в чем наши мнения совпадают.

При том количестве противоречий, которое я признал, пожалуй, было бы разумным рассказать о мандате доверия, дающем мне право отобрать эти факты, а также вступить в спор в случае противоречий. (В начале книги есть моя не очень серьезная биография, поэтому здесь я буду краток.) Я работаю в области технологии программирования уже более 45 лет, в основном в качестве технического специалиста и исследователя. Я написал 25 книг и более 75 профессиональных статей на эту тему. Я регулярно публикуюсь в трех лидирующих журналах отрасли: в *Communications of the ACM* – колонка «The Practical Programmer» (Практикующий программист), в *IEEE Software* – «The Loyal Opposition» (Лояльная оппозиция) и в *SIGMIS DATABASE* издательства ACM – «Through a Glass, Darkly» (Через тусклое стекло). Я известен как человек, придерживающийся своего собственного оригинального мнения, и у меня, в подтверждение моих слов, есть знак «Главного Брюзги практического программирования»! Вы можете рассчитывать на меня, если надо оспорить неоспоримое и, как я сказал ранее, пронзить критикой несколько «священных коров».

Я хотел бы еще кое-что добавить по поводу этих фактов. Я уже говорил, что тщательно подбирал их по критерию фундаментальности для нашей отрасли. Но если говорить о том, сколько из них действительно забыты, то почти все эти сведения нам не удается использовать в своей практике. Заявления менеджеров, руководящих разработчиками, показывают, что либо факты эти забыты, либо о многих из них никто никогда не слышал. Не знают о них и разработчики, чей мир слишком стеснен этим незнанием. Исследователи отстаивают убеждения, которые они сами сочли бы абсурдными, если бы дали себе труд задуматься. Я всерьез полагаю, что те из вас, кто захочет продолжить чтение книги, получат возможность узнать (или вспомнить) многое.

А сейчас, прежде чем оставить вас наедине с фактами, хочу очертить некоторые важные ожидаемые результаты. Представляя факты, я помимо этого часто определяю проблемы отрасли. В мои намерения здесь не входит предлагать решения этих проблем. Эта книга отвечает на вопрос «Что?» а не «Как?». Здесь для меня важно вытащить эти факты обратно на поверхность, где их можно свободно обсуждать и достичь прогресса в практическом следовании им. Я считаю, что это достаточно важная цель, чтобы не нивелировать ее, уводя разговор в сторону решений. Эти решения проблем нередко можно найти в книгах и статьях: учебниках и профессиональных книгах по технологии программирования, ведущих журналах отрасли и популярных компьютерных журналах (хотя многие из последних содержат смесь полезной информации с полным невежеством).

Чтобы помочь преодолеть этот путь, я представляю факты в следующем порядке:

- Сначала я *обсуждаю* факт.
- Затем я представляю *полемику* (если она есть), касающуюся данного факта.
- И под конец я представляю *источники* информации, относящейся к данному факту, библиографию по основным и вспомогательным сведениям. Многие из этих источников стали классикой по стандартам технологии программирования (это и есть часто забываемые факты). От многих веет свежестью завтрашнего утра. О некоторых можно сказать и то и другое.

Я объединил свои 55 фактов в несколько категорий. Вот они:

- О менеджменте
- О жизненном цикле
- О качестве
- Об исследованиях

Заблуждения объединены аналогичным образом:

- О менеджменте
- О жизненном цикле
- Об обучении

Все, достаточно приготовлений! Я надеюсь, вам понравятся факты и заблуждения, представленные мною здесь. И, что еще важнее, я надеюсь, что вы найдете их полезными.

Роберт Л. Гласс
Лето 2002 г.

3

О качестве

Качество – нечеткий термин. Это особенно справедливо применительно к индустрии ПО. Неточность смысла термина является главным предметом замечательной книги «Zen and the Art of Motorcycle Maintenance» [Pirsig, 1974].[1] Ее главный герой сошел с ума в попытках постичь действительное значение слова и отыскать его приемлемое определение!

Как бы самодовольно мы ни смотрели на его сумасшествие (ведь никто из *нас* не спятит по такому поводу), в индустрии ПО все обстоит немногим лучше. О качестве мы говорим так: «Когда я его увижу, я пойму, что это оно». Но на самом деле нет ни приемлемого определения качества программного продукта, ни согласия по поводу того, на ком лежит ответственность за это качество. И даже если мы найдем определение, которое всех устроит, нам еще придется придумать, как измерить достигнутый уровень качества для любого программного продукта. Рассмотрим каждое из этих соображений по очереди.

Нет приемлемого определения? В нашей области по поводу понятия качества существуют чудовищные разногласия. Хуже того, есть специалисты, уверенные, что никаких разногласий нет, но отстаивающие абсолютно неправильное определение. В Факте 46 я привожу определение, которое предпочитаю сам (качество есть совокупность семи свойств), а затем – в Факте 47 – список определений, неправильных с моей точки зрения. На всякий случай предупреждаю, что вы можете со мной не соглашаться.

Кто отвечает за качество? В большинстве книг и курсов, посвященных качеству программных продуктов, говорится, что обеспечение качества – это задача менеджеров. Но если заглянуть вперед и посмотреть на мое

[1] Роберт Пирсиг «Дзен и искусство ухода за мотоциклом». – Пер. с англ. – СПб.: Симпозиум, 2002 г.

определение, опирающееся на семь свойств, образующих качество, то можно увидеть кое-что, имеющее прямое отношение к сугубо техническим аспектам. Одно из них, *модифицируемость*, подразумевает, что разработчик умеет спроектировать ПО таким образом, что впоследствии модификация не составит труда. *Надежность* означает, что методы создания ПО обеспечивают сведение к минимуму вероятности проникновения в него ошибок, а также то, что в процессе устранения ошибок будет задействовано столько многофункциональных инструментов, сколько потребуется. *Переносимость* – свойство ПО, спроектированного таким образом, что его с минимальными затратами можно перенести с одной платформы на другую. Эти и многие другие признаки качества имеют сугубо техническую природу, для их воплощения нужны глубокие знания в области программирования. Работа менеджера состоит отнюдь не в том, чтобы принимать на себя ответственность за достижение качества, а в том, чтобы предоставить специалистам условия для работы, после чего не мешать им.

Почему мы не можем измерить качество? Потому что не только качество с трудом поддается определению, но то же самое можно сказать и о свойствах, перечисленных в Факте 46. Практически невозможно выразить в виде числа понятность, модифицируемость, тестируемость или бóльшую часть других признаков качества. Да, мы *можем* выразить в числах надежность и, до некоторой степени, эффективность, но ужасно скользкий склон, ведущий к измеряемости качества, не становится от этого менее скользким. Сколько-то лет назад Министерство обороны США выделило средства на измерение всех этих «-ностей» [Bowen, Wigle and Tsai, 1985]. Итоговый трехтомный отчет содержал массу таблиц, подлежащих заполнению (на каждую уходило от 4 до 25 часов), и технологических карт, подлежащих выполнению. Увы, когда дым рассеялся, оказалось, что квантификация качества не стала намного ближе по сравнению с началом исследований.

Итак, какие цели я преследую в этом разделе, посвященном качеству?

- Стараюсь исчерпывающе объяснить, что такое качество и чем оно не является.
- Даю обзор некоторых аспектов надежности, например описываю ошибки и тех, кто их делает. В частности, возвращаюсь к некоторым вопросам, сопутствовавшим уже рассмотренным фактам, относившимся к такой стадии жизненного цикла, как удаление ошибок, и возвожу часть этих второстепенных вопросов в ранг полноправных, самодостаточных фактов.

- Рассматриваю некоторые аспекты эффективности. Как мы увидим из следующих ниже фактов, если эффективность имеет значение, то значение это по-настоящему велико. Некоторые факты об эффективности известны уже десятки лет, и они заслуживают того, чтобы здесь их подлинный статус был восстановлен.
- Наблюдательные читатели могут заметить, что из семи признаков качества я выделил для дальнейшего обсуждения лишь два – надежность и эффективность. Это не должно уменьшить ваше внимание к остальным. Дело в том, что к этим двум имеют отношение факты значительные, не только принципиально важные, но и часто забываемые (что в конце концов и есть тема этой книги).

Источники

К двум источникам, указанным в разделе «Ссылки», добавьте этот:

- Glass, Robert L. 1992. *Building Quality Software*. Englewood Cliffs, NJ: Prentice-Hall. In Section 3.9.1, State of the Theory. Эта книга содержит анализ упомянутого выше отчета Министерства обороны США.

Ссылки

- Bowen, Thomas P., Gary B. Wigle, and Jay T. Tsai. 1985. «*Specifications of Software Quality Metrics*». RADC-TR-85-37, Feb.
- Pirsig, Robert M. 1974. *Zen and the Art of Motorcycle Maintenance*. New York: Morrow.

Качество

Факт 46 **Качество есть совокупность свойств.**

Обсуждение

Качество программного обеспечения можно определить массой способов. Здесь я хочу представить определение, прошедшее самое долгое испытание временем.

Под качеством в индустрии ПО понимают совокупность семи свойств, которыми должен обладать программный продукт: переносимости (portability), надежности (reliability), эффективности (efficiency), удобства в использовании (usability, или учета человеческого фактора), тестируемости (testability), понятности (understandability) и модифицируемости (modifiability). Разные специалисты дают этим свойствам не совсем одинаковые названия, но данный список принят подавляющим большинством и существует почти тридцать лет.

Каков же смысл этих свойств?

1. Переносимость означает, что программный продукт можно без труда перенести на другую платформу.
2. Надежность – это свойство программного продукта надлежащим образом выполнять свои функции.
3. Под эффективностью программного продукта понимают экономное расходование им времени и занимаемого места.
4. Принятие в расчет человеческого фактора (что называют также словом «юзабилити») подразумевает, что с программным продуктом легко и удобно работать.
5. Тестируемость ПО есть не что иное, как свойство, характеризующее легкость его тестирования.
6. Понятность ПО – это свойство, характеризующее, насколько легко (или трудно) специалисту, сопровождающему программный продукт, понять его работу.
7. Модифицируемым называют ПО, изменение которого не вызывает трудностей.

Порядок перечисления этих признаков качества не соответствует каким-либо приоритетам. Да это и нельзя сделать каким-либо эффективным способом. Другими словами, нет общепринятой, корректной последовательности, в какой надо было бы пытаться обеспечить их наличие у ПО. Однако нельзя также сказать, что их не следует упорядочивать. Жизненно важно установить такую последовательность с самого начала для каждого отдельно взятого проекта. Так, если программный продукт создается для рынка, где он будет эксплуатироваться на разных платформах, то переносимость расположится если и не в самой голове списка, то где-то неподалеку от нее. Если от успешного исхода операций, которые выполняет инструмент, управляемый программно, зависят человеческие жизни, то первой в списке свойств ПО должна стоять надежность. Если предполагается, что продукту предстоит долгая, насыщенная жизнь, то весьма вероятно,

что ближе к началу списка окажутся свойства, затрагивающие сопровождение, – понятность и модифицируемость (интересно отметить, что два свойства из семи непосредственно относятся к сопровождению). Если же продукт будет работать в условиях дефицита ресурсов, то в верхней части списка мы увидим эффективность.

Например, обычные приоритеты для среднестатистического продукта могли бы быть такими:

1. Надежность (если продукт не работает надлежащим образом, то остальные его свойства не имеют большого значения).
2. Учет человеческого фактора (огромное внимание, уделяемое графическим интерфейсам пользователя в настоящее время, лучше всяких слов говорит о большом значении удобства работы с продуктом).
3. Понятность и модифицируемость (любое ПО, стоящее затраченного на него труда, вероятно, будет долго сопровождаться и поддерживаться).
4. Эффективность (сам удивлен, что поместил это свойство на столь низкую позицию; для некоторых приложений оно будет на первом месте).
5. Тестируемость (предпоследнее свойство, но это не уменьшает его важность, поскольку оно может самым прямым путем привести к надежности ПО, а надежность я поставил на первое место).
6. Переносимость (для многих приложений это свойство вообще не имеет значения, а для других может иметь первостепенную важность).

Не удивляйтесь, если приведенный мною порядок не совпадет с вашим. Когда я писал свою первую книгу о качестве ПО [Glass, 1992], один из рецензентов постоянно пытался изменить порядок, который избирал я (избирал случайно). Он руководствовался при этом своей системой приоритетов (которая, возможно, случайно сильно отличалась от моей). Полагаю, что в попытках создать обобщенную иерархию признаков качества есть что-то от хорошего проектирования ПО: если два программиста приходят к согласию, то они, вероятно, составляют большинство.

Полемика

С этим фактом связано несколько спорных моментов, проистекающих из следующих вопросов:

1. Корректно ли это определение качества?
2. Правильный ли это список свойств?
3. Существует ли правильная последовательность признаков качества?

Что касается пункта 1, то надо сказать, что многие в индустрии ПО (в том числе некоторые эксперты) вошли в лагерь, объединившийся под девизом «Это неправильное определение». Большинство из них придерживается одного из определений, представленных мною в Факте 47. Когда мы будем обсуждать этот факт, я расскажу, почему я думаю, что они просто ошибаются.

Перейдем к пункту 2. В индустрии ПО не все согласны с моим списком признаков качества. Один специалист, например, очень возражает против присутствия в нем переносимости – на том основании, что для продуктов из других отраслей производства она не является признаком качества. Аргумент интересный и, я бы сказал, ошибочный. Не следует думать, что набор признаков качества для разных отраслей универсален в большей степени, чем их приоритеты универсальны по отношению к проектам. Так, очень важным признаком качества для автомобилей являются параметры тюнинга и внешний лоск. Но для многих продуктов, в том числе программных, он не имеет значения. Есть и другие, кто просто придумывает разные названия для признаков качества из приведенного мной списка. К этому я отношусь намного легче. Мне неважно, как их называют, до тех пор пока в определение качества ПО включаются понятия, стоящие за этими названиями.

Ну а пункт 3 мы уже обсуждали. Я утверждаю, что не существует корректного универсального порядка, в котором следует располагать признаки качества ПО, и споры об этом смутно напоминают споры о том, сколько ангелов могут танцевать на острие булавки.

Источники

Самое первое и самое известное упоминание об этом определении качества, основанном на признаках, можно найти в работе Барри Боэма (Barry Boehm).

➥ Boehm, Barry W., et al. 1978. *Characteristics of Software Quality.* Amsterdam: North-Holland.

Исследование Боэма получило наилучшее, на мой взгляд, развитие в работе, упомянутой ниже в разделе «Ссылки». А еще раньше (не намного) определение качества ПО, основанное на признаках, встретилось в статье

➥ McCall, J., P. Richards, and G. Walters. 1977. «Factors in Software Quality». NTIS AD-A049-015, 015, 055, Nov.

Ссылки

➥ Glass, Robert L. 1992. *Building Quality Software*. Englewood Cliffs, NJ: Prentice-Hall.

Факт 47 **Качество не определяется удовлетворением пользователя, соответствием требованиям заказчика, приемлемостью цены и соблюдением сроков сдачи продукта или надежностью.**

Обсуждение

Для качества ПО предложено такое количество разных определений, что иногда я впадаю в отчаяние. Причина моего отчаяния в том, что очень многие из этих определений, несомненно, неправильны, но их поборники, в чем тоже нет сомнений, считают себя правыми.

В этом Факте содержится четыре таких определения. Каждое из них весьма привлекательно, каждое говорит о чем-то, что имеет значение для индустрии ПО. Но я утверждаю, что ни одно из них нельзя считать корректным.

Эти определения мне не нравились, но я долго не мог точно сказать, чем именно. И не смог, пока не услышал, как докладчик из Computer Sciences Corp. на одной из конференций связал все термины в одну формулу. Вот она:

Удовлетворение пользователя = Выполнение требований
+ своевременная поставка
+ приемлемая стоимость
+ качественный продукт

Весьма наглядное определение удовлетворения пользователя. Пользователь будет удовлетворен, если вовремя получит продукт, соответствующий запросам, имеющий приемлемое качество и не стоящий целого состояния. Но если пристально проанализировать эту формулу, то окажется, что все ее важные составляющие – разные и не смешиваются друг с другом. Обратите внимание, что одной из составляющих является качество.

И это говорит о том, я бы особенно подчеркнул, что качество явственно отличается от всех остальных составляющих формулы.

Заметьте, что все слагаемые этой формулы очень важны. Говоря, что качество – это не то же самое, что все остальные составляющие, мы не умаляем значение последних. Мы только говорим, что качество – это вообще нечто другое. Соответствие требованиям, соблюдение сроков поставки и приемлемость цены существенно важны, но они не имеют отношения к качеству. Удовлетворенность пользователя имеет отношение к качеству, но не только к нему, а и к некоторым другим очень важным аспектам.

Это проясняет ситуацию с большинством из того, чем качество не является (в контексте этого факта). Но одна составляющая остается – это надежность. Многие специалисты приравнивают качество ПО к наличию в нем ошибок (или, следовательно, к их отсутствию). Но, как мы видим из Факта 46, качество имеет прямое отношение к ошибкам – именно их имеют в виду, говоря о «надежности», но последняя объединяет намного больше понятий.

Специалисты, приравнивающие качество к отсутствию ошибок, часто все прекрасно понимают. Во время обсуждения они соглашаются со всеми признаками качества, но сразу после этого можно услышать, как они говорят об ошибках так, будто качество – это только их отсутствие. Так заманчиво приравнять качество к надежности: ведь последняя так важна для первого (с неохотой я отвел ей первое место в Факте 46) – но тогда со сцены уходят некоторые другие, чрезвычайно важные признаки.

Полемика

Этот факт сам по себе – сплошная полемика. Не прекращаются разговоры о том, что качество – это удовлетворение пользователя, или соответствие требованиям, или предварительным оценкам (я вообще не понимаю, как попал в список этот пункт, не имеющий, по моему убеждению, никакого отношения к качеству), или надежность. И эти разговоры подкреплены сильной убежденностью, не уменьшающей, впрочем, неправоты убежденных.

Источники

Не привожу источников для этих ошибочных альтернативных взглядов на качество ПО, поскольку а) тем самым я бы укрепил эти ошибочные взгляды и б) мне пришлось бы в этом случае опровергать мнение людей,

имена которых могут быть вам известны. Здесь для всех нас очень важно принять корректное определение качества и отбросить ошибочное. Пока мы не сделаем этого, нам будет трудно в сколько-нибудь конструктивном ключе говорить о качестве ПО.

Надежность

Факт 48 **Есть такие ошибки, к которым предрасположено большинство программистов.**

Обсуждение

Наверное, никого не должно удивлять, что какие-то ошибки в программных продуктах встречаются чаще, чем другие. Всякий, кто когда-либо принимал участие в инспекции исходного кода, наверное, помнит, как кто-то из проверяющих приговаривал: «Так-так, опять одна из этих пресловутых ошибок». Дело в том, что человек подвержен совершению ошибок определенного рода. Индексирование с ошибкой на единицу (off-by-one). Ссылка на переменную, предшествующая ее определению. Пропуск тонких деталей проектирования. Отсутствие инициализации совместно используемых переменных. Отсутствие важного условия в списке условий.

Странно, но не многие исследователи удостоили этот предмет своим вниманием. Помимо собственного опыта, для меня источником явилась работа исследователя из Германии, опубликованная в материалах малоизвестной конференции в Бременхафене [Gramms, 1987]. Он назвал эти ошибки «систематическими» и сказал, что они проистекают из «ловушек мышления». Это не совсем понятно, поскольку можно ожидать, что эти ошибки попадут в число тех, на которых будут сосредоточены методы устранения ошибок. Например, они могли бы быть внесены в контрольные списки инспекций. Можно было бы создать инструментальные средства, изолирующие и идентифицирующие эти ошибки. Можно было бы непосредственно организовать тесты так, чтобы перехватывать их. Исследователи могли бы изучить дополнительные способы их обнаружения и предотвращения.

У систематических, или общих, ошибок есть и еще одна грань. Среди понятий отказоустойчивого программирования (так называют создание программ, пытающихся перехватывать собственные ошибки, когда они случаются) есть так называемое N-вариантное программирование. В осно-

ве последнего лежит предположение, что в N различных решениях задачи, выполненных N различными командами программистов, вряд ли будут повторяться одни и те же ошибки. Если они будут работать совместно друг с другом, то можно будет идентифицировать и отбросить как ошибочный любой результат, получаемый по одному из вариантов, если он не совпадет с результатами, полученными по остальным. Но гипотеза о систематических ошибках предполагает, что более чем в одном из N вариантов решения есть одна и та же ошибка, что снижает силу N-вариантного подхода. (И это серьезное препятствие в тех областях, где критически важные задачи должны иметь сверхнадежные решения, например в системах, обеспечивающих воздушное и железнодорожное сообщение.)

Полемика

Феномен, описанный в этом факте, способен удивить не многих, имеющих отношение к индустрии ПО, но он не получил широкого признания и поэтому не может вызвать какую бы то ни было полемику.

Источник

Я знаю только конференцию, которую проводило Германское компьютерное общество (German Computing Society), и которая упомянута в разделе «Ссылки». А вот моя работа, куда я включил некоторые из находок Граммза (Gramms):

- Glass, Robert L. 1991. «*Some Thoughts on Software Errors*». In *Software Conflict*. Englewood Cliffs, NJ: Yourdon Press. Перепечатано с заметок, использованных при подготовке упомянутого выше выступления на конференции Германского компьютерного общества.

Ссылки

- Gramms, Timm. 1987. В материалах обсуждаются «систематические ошибки» и «ловушки мышления». Заметки группы технических исследований отказоустойчивых вычислительных систем Германского компьютерного общества (German Computing Society Technical Interest Group on Fault-Tolerant Computing Systems, Bremerhaven, West Germany, Dec).

Факт 49 Ошибки имеют тенденцию образовывать скопления.

Обсуждение

Ознакомьтесь с мнениями о том, где и в каких количествах обнаруживаются программные ошибки.

- «Половина ошибок обнаруживается в 15% модулей» ([Davis, 1995], цитата из работы Эдреса [Endres, 1975]).
- «80% всех ошибок обнаруживаются всего в 2% (sic) модулей» ([Davis, 1995], цитата из [Weinberg, 1992]). А если посмотреть следующую цитату, то можно подумать, уж не опечаткой ли были эти 2%.
- «Примерно 80% ошибок находятся в 20% модулей, а примерно половина модулей не содержит ошибок» [Boehm and Basili, 2001].

Какими бы ни были действительные значения, очевидно, что ошибки чаще всего образуют скопления в программных продуктах. Заметьте, что этот факт известен уже несколько десятилетий – работа Эндреса датирована 1975 годом.

Почему ошибки группируются именно так? Может быть, некоторые части программ заметно сложнее других, и эта сложность приводит к ошибкам? (Таково мое мнение.) Может быть, написание программ поручается нескольким программистам, а некоторые из них совершают ошибки чаще, обнаруживая их реже? (Конечно, и это возможно, учитывая индивидуальные отличия, о которых мы говорили в Факте 2.)

В чем же смысл этого конкретного факта? Найдя в некотором программном модуле больше ошибок, чем ожидали, продолжайте поиски. Весьма вероятно, что их найдется еще больше.

Полемика

Данные, приведенные здесь, достаточно ясны и собирались достаточно долго, и я не знаю о каких бы то ни было дискуссиях по этому Факту.

Источники

Источники, содержащие информацию, подтверждающую данный факт, перечислены в разделе «Ссылки».

Ссылки

- Boehm, Barry, and Victor R. Basili. 2001. *«Software Defect Reduction Top 10 List». IEEE Computer*, Jan.
- Davis, Alan M. 1995. *201 Principles of Software Development*. New York: McGraw-Hill. Principle 114.
- Endres, A. 1975. *«An Analysis of Errors and Their Causes in System Programs». IEEE Transactions on Software Engineering*, June.
- Weinberg, Gerald. 1992. *Quality Software Management: Systems Thinking*. Vol. 1, section 13.2.3. New York: Dorset House.

Факт 50 **Для устранения ошибок еще не выработан какой-то один, лучший подход.**

Обсуждение

Избыточность – вот что надо обсуждать! Это я доказывал несколько раз, некоторое количество Фактов тому назад. Здесь я повторяюсь, потому что данный предмет заслуживает статуса самостоятельного факта, рассматриваемого отдельно от других.

Каково же тайное значение этого факта? Не существует серебряной пули, способной сразить все ошибки. И не похоже, что таковая когда-нибудь появится. Тестирование не дает гарантии, какого бы оно ни было вида. Не дают гарантии инспекции и экспертные проверки, и при этом неважно, кто давал им определения. Доказательство корректности программы (если вы верите в подобные штуки) не дает гарантии. Каким бы ценным качеством ни была отказоустойчивость, она не дает гарантии. И каким бы ни был ваш любимый инструмент устранения ошибок, он тоже не дает гарантии.

Полемика

В данном случае полемику в основном провоцируют крикуны. Приверженцы серебряных пуль не прекратят производство мыльных пузырей рекламы любых продаваемых ими технологий, в том числе и способов устранения ошибок. Эти приверженцы закоснели в своей неправоте, но это не помешает им раздувать тот же самый огонь и в будущем.

Источник

Данный факт составляет одну из главных тем работы

- Glass, Robert L. 1992. *Building Quality Software*. Englewood Cliffs, NJ: Prentice-Hall.

Факт 51 **От ошибок никуда не деться. Цель состоит в том, чтобы избежать критических ошибок или свести их к минимуму.**

Обсуждение

Больше избыточности! Повторю еще раз: я упоминаю об этом здесь потому, что избыточность заслуживает возведения в ранг самостоятельного факта, к которому мы не должны добираться попутно, влекомые к нему другими фактами.

В программах всегда будут необнаруженные дефекты, даже после самого тщательного из процессов устранения ошибок. Мы должны свести к минимуму количество и особенно серьезность этих остаточных дефектов.

Принципиально важно, говоря об ошибках в ПО, ввести понятие критичности ошибки (error severity). Критические ошибки в программных продуктах должны быть устранены. Хорошо было бы устранить и все остальные ошибки (например, ошибки документирования, ошибки избыточного кода, ошибки недоступных путей, алгоритмические ошибки, не влияющие на вычисления), но это не всегда необходимо.

Полемика

Все согласны, что в программах есть общие необнаруженные ошибки. Но очень много спорят о том, должно ли сохраниться такое положение дел. Реалисты (их можно было бы назвать пессимистами или даже апологетами) уверены, что ситуация не изменится (из-за всех этих уже рассмотренных нами сложностей). Оптимисты (их можно было бы назвать мечтателями или даже адвокатами) не сомневаются, что мечта о ПО, свободном от ошибок, осуществима, и надо лишь ввести весь процесс в достаточно строгие рамки.

Одно недавнее исследование [Smidts, Huang and Widmaier, 2002] проливает свет на важные аспекты данного вопроса. Две команды, следуя двум различным подходам (традиционному, соответствующему уровню 4 модели CMM, и авангардному, в котором применялись формальные методы), не смогли создать достаточно простой продукт, который бы удовлетворял заданному уровню надежности в 98% (хотя к обеим командам предъявлялись довольно-таки щадящие требования в смысле стоимости и сроков сдачи).

К настоящему моменту вы, вероятно, уже выбрали, на чьей стороне ваши симпатии в этой полемике. Я же оставляю эту тему.

Источник

Данный факт является одним из главных предметов книги.

➥ Glass, Robert L. 1992. *Building Quality Software*. Englewood Cliffs, NJ: Prentice-Hall.

Приведем некоторые интересные мысли, относящиеся к рассмотренному вопросу.

О необнаруженных ошибках:

- «Строгое регламентирование выполняемых работ способно снизить количество ошибок до 75%» [Boehm and Basili, 2001].
- «Примерно 40–50% пользовательских программ содержат нетривиальные ошибки» [Boehm and Basili, 2001]. (Заметьте, что данное высказывание имеет отношение и к серьезности ошибок.)
- «Все ошибки обнаружить невозможно» [Kaner, Bach and Pettichord, 2002].

О серьезности ошибок:

- «Почти 90% простоев вызвано максимум 10% ошибок» [Boehm and Basili, 2001].

Ссылки

➥ Boehm, Barry, and Victor R. Basili. 2001. «Software Defect Reduction Top 10 List». *IEEE Computer*, Jan.

➥ Kaner, Cam, James Bach, and Bret Pettichord. 2002. *Lessons Learned in Software Testing*. Lessons 9, 10. New York: John Wiley & Sons.

➥ Smidts, Carol, Xin Huang, and James C. Widmaier. 2002. «Producing Reliable Software: An Experiment». *Journal of Systems and Software*, Apr.

Эффективность

Факт 52 **Эффективность больше зависит от качественного проектирования приложения, чем от качественного программирования.**

Обсуждение

Программисты, никогда не теряющие оптимизма, долгие годы не сомневаются, что знают способ написать эффективную программу. Поэтому в частности до сих пор жив такой язык, как ассемблер. Ему посвящен Факт 53, а здесь я хочу принести в жертву священную корову программирования, код, отдав предпочтение проектированию.

Для начала необходимо подумать, каковы истоки недостаточной эффективности программного продукта, – тогда данный факт приобретет смысл. В числе прочих источников неэффективности назовем внешний ввод/вывод (I/O) (например, медленный доступ к данным), неуклюжие интерфейсы (не вызванные необходимостью или удаленные вызовы процедур), а также потери времени, обусловленные внутренними причинами (например, логические циклы, ведущие в никуда).

Начнем с недостатков ввода/вывода. На выборку и размещение данных на внешних носителях компьютеры затрачивают неизмеримо больше времени, чем на любые другие выполняемые ими операции. И цент, сэкономленный на проектировании операций ввода/вывода, оборачивается долларом, заработанным на ускорении работы приложения. Выбор форматов данных богат, и он определяет эффективность конечной программы в большей степени, чем любой другой выбор, сделанный программистом. Я убеждал университетских преподавателей по вычислительной технике, что главная задача читаемых ими базовых курсов по структурам данных и файлов состоит в том, чтобы разъяснить, какие структуры и в каких приложениях наиболее эффективны. Зачем, в конце концов, при наличии простого последовательного доступа и даже хешированных данных понадобилось придумывать связные списки, деревья, индексирование и т. д? И так ли уж нам необходимо кэширование данных, снижающее эффективность логики программы единственно ради более эффективной работы с данными? Затем, говорил я им, что структуры данных представляют собой компромисс между увеличением сложности их архитектуры и повышением эффективности доступа к их содержимому. (Некоторые ученые убеждены, что

проблема эффективности канула в прошлое, и видят в структурах данных лишь интересные способы организации данных, не больше.)

Поэтому во время проектирования мы очень глубоко задумываемся над выбором правильной структуры данных. Или структуры файлов. Или архитектуры базы данных. Зачем тратить массу сил, раньше времени программируя никуда не годную схему доступа к данным?

Неэффективность интерфейса или логики ничтожна в сравнении с неэффективностью ввода/вывода. Хотя, конечно, программист может заставить колеса программной логики крутиться с необычно низкой скоростью, если неудачно реализует циклическую обработку данных. (Существуют даже «бесконечные циклы», работа которых не завершается никогда.) Возможно, самые тяжкие преступления лежат на совести итеративных вычислительных алгоритмов. Медленная или нулевая сходимость алгоритма может привести к чудовищным затратам процессорного времени. В непосредственной близости, но чуть позади, располагаются неэффективные методы доступа к структурам данных (данным не обязательно находиться на внешнем носителе, чтобы доступ к ним был связан с трудностями). Повторюсь: некоторое внимание, уделенное рациональному алгоритму на этапе проектирования, может дать намного более впечатляющий результат, чем внешне эффектный программный код.

Каков же заключительный вывод? Если программный продукт должен быть эффективным, то заботиться об этом надо на ранних стадиях его жизненного цикла – если говорить точнее, то до начала программирования.

Полемика

Для некоторых рьяных программистов написание кода – самая важная составляющая процесса создания ПО, и они считают, что чем быстрее оно начинается, тем лучше. Для сторонников этого лагеря проектирование есть всего лишь нечто такое, что задерживает работу, направленную на полное решение проблемы.

Но они обращаются лишь к сравнительно простым проблемам, и поэтому а) их, вероятно, никогда не удастся убедить в обратном и б) в том, что они делают, не может быть ничего уж очень неправильного. Но большая сложность и не нужна для того, чтобы этот подход, основанный на таком скупом проектировании и поспешном написании кода, потерпел не-

удачу. (Вспомните Факт 21, где говорилось, как скоро сложность задачи вызывает увеличение сложности ее решения.)

Моя мысль сводится к тому, что полемика по поводу этого конкретного факта имеет место между теми, кто считает ценность проектирования незначительной, и теми, кто рассматривает этот этап как жизненно важную прелюдию к собственно написанию кода. Несомненно, что экстремальное программирование, отстаивающее простые подходы к проектированию, позволяющие быстро начать кодирование, добавляет масла в огонь этих споров. Поэтому неудивительно, что в этой технологии такое внимание уделяется «рефакторингу», призванному исправлять несовершенства и ошибки проекта, уже переведенного в программный код.

Источники

Данный факт принадлежит к тем, которые известны уже так давно, что почти невозможно проследить их историю до какого-либо печатного источника. О нем говорится в любой книге, посвященной созданию программных продуктов (а они издаются уже лет тридцать).

Из следующих книг по экстремальному программированию можно понять, почему считается, что надо побыстрее закончить проектирование и приступить к написанию кода и последующему его рефакторингу, исправляющему ошибки, которые были сделаны по ходу этого стремительного броска к коду:

- Beck, Kent. 2000. *eXtreme Programming Explained*. Boston: Addison-Wesley. См. материал по «простому проектированию» и «рефакторингу».[1]

Подробнее всего рефакторинг рассматривается в книге

- Fowler, Martin. 1999. *Refactoring: Improving the Design of Existing Code*. Reading, MA: Addison-Wesley.[2]

---

[1] Кент Бек «Экстремальное программирование. Библиотека программиста». – Пер. с англ. – Спб.: Питер, 2002.

[2] Мартин Фаулер, Бек К., Брант Д., Робертс Д., Апдайк У. «Рефакторинг: улучшение существующего кода». – Пер. с англ. – СПб.: Символ-Плюс, 2002.

Факт 53 **Эффективность кода на языке высокого уровня, компилированного с соответствующими параметрами оптимизации, может достигать 90% эффективности функционально сравнимого ассемблерного кода. А в некоторых сложных современных архитектурах она может быть даже выше.**

Обсуждение

В индустрии ПО споры о сравнительных достоинствах языка ассемблера и языков высокого уровня (ЯВУ) ведутся уже долгие годы. На что только не идут приверженцы каждого из лагерей в попытках доказать превосходство своей точки зрения на примере конкретного проекта. Почти так же давно известны данные, которые позволяют разрешить этот спор. В конце раздела приведен список источников, в котором можно найти исследования, проведенные в 70-х годах прошлого столетия. Поэтому, хотя спор языков высокого уровня с ассемблером примерно так же свеж, как и сфера приложений для самообучения (где, кажется, даже самые несгибаемые сторонники ассемблера сдают свои позиции), мудрость, накопленная в индустрии ПО, вероятно, окажется достаточной, чтобы дебаты прекратились.

Шквал исследований, предпринятых в середине 1970-х, позволил получить исчерпывающие данные. Шквал был вызван тем, что этот спор приобрел критическое значение в развивающейся сфере приложений для авионики (ПО, создаваемого для управления электронными приборами воздушных и космических летательных аппаратов). А вот какие при этом были сделаны открытия, безупречно подытоженные в работе [Rubey, 1978]: «Почти во всех докладах сообщается ... о том, что неэффективность приложений для авионики, написанных на языках высокого уровня, составляет 10–20%.» (Далее в этой работе отмечалось, что применение оптимизирующих компиляторов позволяет увеличить эффективность кода еще минимум на 10% и что тонкая настройка программы на языке высокого уровня после ее написания (этот процесс нетрудно сравнить с доводкой программы на языке ассемблера) может добавить еще от 2 до 5%.)

Так почему же этот спор не утихал все эти десятилетия? Дело в том, что, несмотря на все несомненные преимущества языков высокого уровня, бывают случаи, когда и вправду следует предпочесть ассемблер. Другими словами, преимущества языков высокого уровня сильно зависят от задачи – для некоторых задач написать эффективную программу на таком языке намного труднее , чем для других.

Каковы же преимущества ЯВУ? Кажется, что называть их почти не имеет смысла, т. к. они хорошо известны и общепризнанны, и тем не менее.

- Решение задачи на языке высокого уровня занимает намного меньше строк программного кода, чем на языке ассемблера, благодаря чему разница в производительности труда программиста получается впечатляющей.
- Код на этих языках обеспечивает возможность элегантной и искусной работы в трудных и чреватых ошибками областях, таких как манипуляции с регистрами.
- Высокоуровневый код в целом характеризуется переносимостью, программы на ассемблере не переносимы.
- Высокоуровневый код легче поддерживать (понимать и модифицировать).
- Высокоуровневый код легче поддается тонкой настройке, призванной повысить, там, где это необходимо, его эффективность.
- Для написания высокоуровневого кода программисту нужна менее высокая квалификация.
- Высокоуровневые компиляторы способны оптимизировать использование современных архитектур, например конвейеризованных или задействующих кэш-память.

В чем преимущества ассемблера?

- Операторы языков высокого уровня не всегда согласуются с функциональными особенностями аппаратного обеспечения, а код ассемблера может использовать преимущества прямого обращения к аппаратным ресурсам, будучи при этом проще своих высокоуровневых аналогов.
- Аналогично на языке высокого уровня не всегда удобно организовывать взаимодействие с функциями ОС, ориентированными на ассемблер.
- Нехватка места в памяти и требования к скорости выполнения иногда предъявляют жесткие требования к эффективности кода.

Удачное исследование, посвященное применению ассемблера в приложениях, работающих в условиях жестких ограничений, сделано в работе [Prentiss, 1977]. В этом проекте исходная программа была написана полностью на языке высокого уровня, а затем – в русле изучения получившихся проблем с эффективностью – 20% кода было переписано на ассемблере. Это соотношение, 80/20, вполне адекватно отражает максимальное коли-

чество ассемблерного кода, в котором может нуждаться любая, даже современная система.

Полемика

Иногда кажется, что споры на эту тему не кончатся вообще! Ассемблер – это технология в некотором смысле соблазнительная – какой «настоящий программист» не хочет по-настоящему близко познакомиться со своим компьютером и с операционной системой? Однако на самом деле этот спор был в очень большой степени разрешен еще в 1970-х. Беда в том, что мало кому из современных программистов известны покрытые пылью исследования в области авионики.

Источники

Источники, материалы которых нашли применение в этом факте, перечислены в разделе «Ссылки».

Ссылки

- Prentiss, Nelson H., Jr. 1977. *«Viking Software Data»*. RADC-TR-77-168. Фрагменты этого исследования, в том числе имеющие отношение к противостоянию языков высокого уровня и ассемблера, были перепечатаны в книге Robert L. Glass, *Modern Programming Practices* (Englewood Cliffs, NJ: Prentice-Hall, 1982).
- Rubey, Raymond. 1978. *«Higher Order Languages for Avionics Software – A Survey, Summary, and Critique»*. Труды NAECON (Национальной конференции по аэронавтике и электронике), перепечатаны в книге Robert L. Glass, ed., *Real-Time Software* (Englewood Cliffs, NJ: Prentice-Hall, 1983). В этой работе можно найти ссылки на исследования сравнительной эффективности ассемблера и языков высокого уровня.

Факт 54 **Существуют компромиссы между оптимизацией по размеру и оптимизацией по скорости. Нередко улучшение первого показателя вызывает ухудшение второго.**

Обсуждение

Может показаться, что любое изменение, после которого программа работает быстрее, сделает ее и более компактной. Однако это не так.

Возьмем для примера что-нибудь банальное, например тригонометрическую функцию. Большинство тригонометрических функций вычисляются алгоритмически, при этом программный код занимает очень немного места, но в силу итеративной природы алгоритма исполняется не быстро. Альтернативный способ реализации тригонометрической функции состоит в том, чтобы встроить в программу таблицу значений и вычислять результат путем интерполяции. Интерполяция выполняется намного быстрее, чем итерации, зато таблица намного больше, чем программный код итеративного алгоритма. (Этому решению было отдано предпочтение при создании (с моим участием) ПО для космической системы, где скорость выполнения кода была неизмеримо важнее его размера.)

В качестве еще одного, более современного примера рассмотрим программы на Java. Код на языке Java компилируется не в машинные инструкции, а в так называемый байт-код. Байт-код намного компактнее эквивалентного машинного кода, и в результате программы на Java эффективны с точки зрения их размера. Но эффективны ли они с точки зрения скорости исполнения? Ни в коем случае! Поскольку байт-код – это не машинный код, он интерпретируется по мере исполнения программы, а интерпретация иногда увеличивает время исполнения в 100 раз.

Таким образом, поиски эффективности сводятся к достижению компромисса. Количество случаев, когда увеличение скорости работы программы сопровождалось уменьшением ее размера, невелико. (В работе [Rubey, 1978]), выполненной в русле исследований эффективности языков высокого уровня, было сделано интересное наблюдение, что легче оценить эффективность кода с точки зрения размера, чем с точки зрения скорости его исполнения. Первую, в общем случае, можно измерить статически, тогда как последнюю приходится измерять динамически. Хорошо это или плохо, но по этой причине иногда легче создать компактное приложение, чем быстродействующее.)

Какой из этого следует вывод? Стремясь создать программный продукт, который обладал бы таким признаком качества, как эффективность, убедитесь, что точно знаете, какая именно эффективность нужна в данном случае.

Полемика

Внимание на этом факте заостряют нечасто, и разногласий с ним связано немного. Бóльшая часть тех, кого интересует эффективность, об этом уже знают. Замечу, однако, что те, кто не интересуется эффективностью, могут совершить весьма серьезные ошибки, не приняв этот факт во внимание.

Источник

Источник информации для этого факта указан в разделе «Ссылка».

Ссылка

➥ Rubey, Raymond. 1978. «Higher Order Languages for Avionics Software – A Survey, Summary, and Critique». Труды NAECON ((Национальной конференции по аэронавтике и электронике). Перепечатано в книге Robert L. Glass, ed., *Real-Time Software* (Englewood Cliffs, NJ: Prentice-Hall, 1983).