ПРОФЕССИОНАЛЬНО

Роберт Гласс

КРЕАТИВНОЕ программирование

2.0

Символ®

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 978-5-93286-152-3, название «Креативное программирование 2.0» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.

software creativity 2.0

Robert L. Glass

d.* BOOKS

ПРОФЕССИОНАЛЬНО

КРЕАТИВНОЕ ПРОГРАММИРОВАНИЕ 2.0

Роберт Гласс

СИМВОЛ®

Санкт-Петербург — Москва
2009

Серия «Профессионально»

Роберт Гласс

КРЕАТИВНОЕ ПРОГРАММИРОВАНИЕ 2.0

Перевод С. Маккавеева

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Выпускающий редактор	*П. Щеголев*
Редактор	*Т. Темкина*
Корректор	*С. Минин*
Верстка	*Д. Орлова*

Гласс Р.

Креативное программирование 2.0. – Пер. с англ. – СПб.: Символ-Плюс, 2009. – 352 с., ил.

ISBN 978-5-93286-152-3

Роберт Гласс исследует важный, но часто упускаемый из виду вопрос о роли творчества в программном инжиниринге и программировании. Почти полувековой личный опыт как разработчика, преподавателя и исследователя помогают автору охватить множество проблем. Что важнее – процесс или продукт? Каково соотношение между «интеллектуальным» и «канцелярским» трудом, а также между теорией и практикой в программировании и как сделать их взаимодействие более эффективным? В каких ситуациях полезнее подход со строгим контролем, а когда лучше свободное экспериментирование?

Первое издание, вышедшее в 1995 году, стало бестселлером. Второе издание подверглось значительной доработке. Том Демарко в своем предисловии сравнивает его со знаменитым «Мифическим человеко-месяцем» Фредерика Брукса. Первоначальный текст был дополнен результатами более чем десятилетних поисков недостающего звена между творчеством и программированием. Со свойственным автору мягким юмором и эрудированностью он дает читателю советы и рекомендации огромной важности, проливает свет на историю становления индустрии разработки ПО, исподволь подводя к мысли о том, что настоящий программист – это не оператор, не наборщик кода, а настоящий Творец.

ISBN 978-5-93286-152-3
ISBN 978-0-9772133-1-3 (англ)

© Издательство Символ-Плюс, 2009

Authorized translation of the English edition © 2006 Developer.* books. This translation is published and sold by permission of Developer.* books, the owner of all rights to publish and sell the same.

Все права на данное издание защищены Законодательством РФ, включая право на полное или частичное воспроизведение в любой форме. Все товарные знаки или зарегистрированные товарные знаки, упоминаемые в настоящем издании, являются собственностью соответствующих фирм.

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Подписано в печать 25.05.2009. Формат 70x90 1/16 . Печать офсетная.
Объем 22 печ. л. Тираж 1000 экз. Заказ №
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Всем, кто, занимаясь программированием, старается изменить мир.

~

«...потому что только тот, кто достаточно безумен,
чтобы надеяться изменить мир, действительно
изменяет его...»

Джек Керуак

Благодарности

Хочу поблагодарить...

...Стива Макконнелла (Steve McConnell, автор «Code Complete»[1] и многих других важных современных книг о разработке программного обеспечения) за активную поддержку и критику...

...Дэна Рида (Dan Read) из developer., который был в высшей степени любезным издателем...*

...Университет Линкопинга в Швеции, где я читал лекции, работая над первоначальным вариантом этой книги, и Университет Гриффита в Австралии, где я читал лекции, работая над ее новой версией.

Роберт Л. Гласс

[1] Стив Макконнелл «Совершенный код. Практическое руководство по разработке программного обеспечения». – Пер. с англ. – СПб.: Питер, 2006.

Другие книги Роберта Гласса

Опубликованы developer.* Books

«Software Conflict 2.0: The Art and Science of Software Engineering», 2006

Опубликованы Computing Trends

«Software 2020», 1998

«Software Folklore», 1991

«Computing Shakeout», 1987

«Computing Catastrophes», 1983

«Software Soliloquies», 1981

«The Second Coming: More Computing Projects Which Failed, with Sue DeNim», 1980

«The Power of Peonage», 1979

«Tales of Computing Folk: Hot Dogs and Mixed Nuts», 1978

«The Universal Elixir, and other Computing Projects Which Failed», 1977

Опубликованы Addison-Wesley/Prentice-Hall/Yourdon Press

«Facts and Fallacies of Software Engineering», 2003[1]

«ComputingFailure.com», 2001

---

[1] Роберт Гласс «Факты и заблуждения профессионального программирования». – Пер. с англ. – СПб.: Символ-Плюс, 2008.

«Computing Calamities», 1999

«Software Runaways», 1998

«An ISO 9000 Approach to Building Quality Software, with Osten Oskarsson», 1996

«Software Creativity», 1995

«Measuring and Motivating Maintenance Programmers, with Jerome B. Landsbaum», 1992

«Building Quality Software», 1992

«Software Conflict», 1991

«Measuring Software Design Quality, with David N. Card», 1990

«Software Communication Skills», 1988

«Real-Time Software», 1984

«Modern Programming Practice–a Report From Industry», 1982

«Software Maintenance Guidebook, with Ronald A. Noiseux», 1981

«Software Reliability Guidebook», 1979

Опубликованы IEEE Computer Society Press

«In the Beginning: Recollections of Software Pioneers», 1998

Оглавление

Предисловие Тома Демарко 13

Предисловие к первому изданию 16

Предисловие ко второму изданию 19

Почему «Креативное программирование»? 21

Часть I. Два типа мышления: исследование творчества в области программирования 25

Введение 26

1. Дисциплина и гибкость 29

Введение 30

1.1. Генри Форд от программирования, пожалуйста, встаньте! 32

1.2. Автоматизация программирования – факт или фикция? 36

1.3. Правда ли, что программистами «невозможно управлять»? 39

1.4. Дисциплина – неприличное слово: рассказ о жизненном цикле разработки программного продукта 43

1.5. Имитация программной разработки 46

1.6. Гибкое программирование (Agile): гибкость достигла зрелости 48

1.7. Странный случай с карандашом корректора 54

1.8. Индекс сложности 56

1.9. «Странная парочка» – дисциплина и творчество 58

2. Формальные методы и эвристики 61

Введение 62
2.1. По поводу одной дискуссии 65
2.2. Программирование без комплекса вины 71
2.3. Формальные методы: драма (успех, провал) 74
2.4. За границами формальных методов 77
2.5. Письма читателей: размышления о формальных методах 79

3. Оптимизация и разумная достаточность 83

Введение 84
3.1. Принцип «лучшее – враг хорошего» и решение задач 86
3.2. Достаточно хорошее программное обеспечение 89
3.3. В защиту методов ad hoc 91
3.4. Михаил Горбачев и продуктивность программирования (!?) 94

4. Количественный и качественный подходы 97

Введение 98
4.1. «Нельзя управлять тем, что нельзя измерить» – так ли это? 101
4.2. Математика и компьютерные науки 103
4.3. Роль интуиции в принятии решений 109
4.4. Тьма ловушек: числа бывают разные 113

5. Процесс или продукт? 117

Введение 118
5.1. Приводит ли хороший процесс к лучшему продукту? 121
5.2. Приводит ли хороший процесс к лучшему продукту? Другая точка зрения 124
5.3. В шаге от славы 126
5.4. Разные мысли по поводу процесса программирования 130
5.5. Процессы и люди: путь к хорошему продукту 132
5.6. Оценка результатов применения модели зрелости процесса (CMM) 134
5.7. Когда лучше ориентироваться на продукт, а когда – на процесс? 137

6. Интеллектуальные и канцелярские задачи 141

Введение 142
6.1. Программы: трудно или легко их делать? 145

6.2. Задачи программирования: интеллектуальные, канцелярские или... творческие? . . . 157
6.3. Почему люди «покупаются» на революционные идеи . . . 170

7. Теория и практика . . . 175

Введение . . . 176
7.1. Временное соотношение теории и практики . . . 179
7.2. Теория против практики – новый взгляд . . . 184
7.3. Теория и практика: тревожный пример . . . 186
7.4. Полет шмеля . . . 188
7.5. Теория и практика: разные жалобы . . . 188
7.6. Области программирования, где практика опережает теорию . . . 191

8. Наука и производство . . . 199

Введение . . . 200
8.1. Дихотомия интересного/полезного . . . 202
8.2. Дихотомия индивидуального/группового . . . 204
8.3. Две фразы из поп-культуры . . . 205
8.4. О понимании, согласии... и формальных методах . . . 205
8.5. Структурированное исследование? (Немного лукавое предложение) . . . 208
8.6. Плавающее соотношение передача/прием . . . 211
8.7. Заседание комиссии, не ориентированной на цель . . . 214
8.8. Точность и актуальность . . . 217

9. Забавность и серьезность . . . 225

Введение . . . 226
9.1. Разрываясь между забавностью и скукой . . . 228
9.2. Open Source: возвращение забавности . . . 232
9.3. Путешествие по странному проекту . . . 235
9.4. Помогите найти! . . . 237

Часть II. Как стимулировать творчество . . . 239

Введение . . . 240

10. Творчество в программирующей организации . . . 243

Введение . . . 244
10.1. Греция и Рим: две очень разные культуры программирования . . . 245

10.2. Контроль и корпоративная культура 248
10.3. Управление и новаторство 250
10.4. Творчество и стратегические информационные системы 253
10.5. Творчество против закона 258

11. Творчество в программных технологиях 261

Введение 262
11.1. Примеры творческого подхода к реализации информационных систем 263
11.2. Творчество и разработка ПО: недостающее звено 267
11.3. Творчество и реальность: конкретный случай 279

12. Исторические даты творчества в программировании 291

Введение 292
12.1. Первая историческая дата 293
12.2. Следующие «серебряные пули» 295
12.3. Творчество и стремление к упрощенности 299

Часть III. Краткий обзор творчества в других областях 305

Введение 306

13. Организационное творчество 309
14. Творческая личность 315
15. Помощь компьютера в творчестве 319
16. Парадоксы творчества 325
17. Так было всегда 329

Часть IV. Итоги и выводы 335

Введение 336

18. Синергическое заключение 337
19. Еще несколько выводов 343

Алфавитный указатель 346

Предисловие Тома Демарко

В индустрии разработки программного обеспечения давно утвердилось собственное разделение на «красные» и «синие» штаты, то есть на сторонников авторитарных и либеральных методов. Одни уповают на «дисциплину» и «контроль», другие воспевают «скорость». Такое раздвоение сильно осложняет сам процесс создания программ, поскольку часто люди не способны услышать друг друга. Как и в случае красных и синих в политике, самые громкие голоса раздаются на крайних флангах, а центр представлен относительно слабо. При этом настоящая работа делается как раз на стыке между авторитаристами и либералами.

Боб Гласс активно пропагандирует компромисс, за что мы все должны быть ему благодарны. В книге «Креативное программирование 2.0» он выдвигает рациональный подход, сочетающий в себе лучшее из предлагаемого обоими лагерями; в его этике характер разработки определяется особенностями конкретного проекта, а не способностью экстремистов того или иного толка навязать свое мнение.

Однажды за ужином мой друг и коллега Шейла Брэйди (Sheila Brady), которая в 1990-х руководила проектами разработки ОС в Apple, довольно резко отозвалась о препятствиях, которые должна преодолеть женщина, чтобы стать руководителем. «Предопределенная модель управления – это патриархат», – сказала она. По ее словам, впервые в жизни мы сталкиваемся с управлением в семье, где боссом является отец, и уже поэтому представляем себе руководителя в чем-то похожим на отца. Естественно, такой руководитель любит командовать, поскольку чаще всего именно так складываются отношения между родителями и детьми. Данный результат – почти идеальный пример альфа-самца (вожака стаи). Женщины, по

наблюдению Шейлы, могут быть выдающимися руководителями, при этом нисколько не напоминая вожаков.

У многих из нас, сегодняшних руководителей организаций, где разрабатывают программное обеспечение, отцы тоже были руководителями. Поэтому наше первое представление об управлении связано с Отцом, главой семьи, но и в более позднем возрасте оно также в какой-то мере определяется Отцом – уже как руководителем той или иной организации. То, что когда-то рассказывал Отец о своей конторе или фабрике, запоминается на всю жизнь и влияет на нашу работу, даже мало соответствуя нынешним реалиям.

Например, мой отец руководил литейным производством. Его героями были Тэйлор («Научная организация труда») и Гилбрет («Анализ движения»). Когда мы с ним говорим об управлении, даже нашей маленькой семье не обойтись без раскола на красных и синих. Стоит упомянуть незаменимых работников, как он выходит из себя: «Если бы я обнаружил, что у меня есть незаменимый работник, то немедленно уволил бы его!» Он вырос в Век индустриализации, а я – в Век информации. И управление в эти две разные эпохи развивалось по двум совершенно разным направлениям.

Не подумайте только, что управление, свойственное Веку информации, просто заменило собой управление Века индустриализации. Остатки мышления индустриальной эпохи крепко сидят в глубинах современной организации. Это понудило нас искать, как священный грааль, новые методы, пытаться систематизировать саму систематизационную работу, с помощью которой мы преобразовывали свои организации. Обычно это приводило к столкновениям. Разумное обсуждение методов нередко тонуло в перебранке:

Ты – ковбой.

А ты – любитель покомандовать.

Хакер!

Фашист!

Дело усугубилось тем, что научное сообщество тоже раскололось на два лагеря. Эмпирики и специалисты по психологии познания постоянно разъясняли, что талант выше методов: некоторые разработчики настолько умелы в своем деле, что остальным при всем желании за ними не угнаться, и им остается только тиражировать методы своих более талантливых коллег. В то же время формалисты утверждали, что *есть* единственный правильный метод разработки программного обеспечения, и он настолько систематичен, что можно убедительно доказать его правильность.

Фактически он настолько систематичен, что на его основе можно организовывать промышленное производство.

Но есть еще реальный мир. И тут с блеском проявляет себя Боб Гласс. В книге «Креативное программирование 2.0» он вдумчиво, методично и убедительно излагает свои мысли. Очень часто он делает это творчески, демонстрируя в книге ту изобретательность, за которую ратует в создании программ. Например, обратите внимание на пример с трапециевидной призмой – это что-то вроде чудесного озарения, неожиданного, впечатляющего и глубокого.

Поскольку для XXI века характерны глобализация, коренные перемены и сильная конкуренция, никто не сомневается в усилении роли творчества. В результате теперь гораздо больше говорят о поддержке творчества. На словах все «за». Едва ли найдется компания, руководство которой не разглагольствовало бы о творчестве и его *огромной* роли. Но часто этим оно и ограничивается. Надеюсь, эта книга поспособствует тому, чтобы поддержка творчества в нашем деле продвинулась хотя бы немного дальше слов.

Том Демарко
Кэмден, Мэн

Предисловие к первому изданию

Книга называется «Креативное программирование». Она написана на основании глубокой личной убежденности автора в том, что:

- создание программного обеспечения – это в первую очередь решение задач;
- решение задач всегда требует творческого подхода;
- решать задачи в программировании очень сложно – возможно, сложнее, чем в других видах деятельности;
- следовательно, решение задач в программировании требует максимально творческого подхода.

Я начал писать эту книгу из других побуждений. Меня утомили и разозлили многочисленные утверждения о том, что задачи разработки ПО решаются с помощью дисциплины, формализма и количественного подхода. Мой 40-летний опыт практического решения задач программирования и 6 лет научной работы, в течение которых я размышлял о связи этого опыта с тем, что пишут о программировании, привели меня к противоположному выводу: задачи разработки ПО решаются с помощью гибкости, творчества и качественного подхода. Разумеется, для крупных проектов, над которыми работает много людей, дисциплина – это неизбежное зло; и все же я считал, что прогресс осуществляют выдающиеся люди, а не выдающиеся технологические процессы.

Но стоило мне приступить к работе над книгой и в поисках материалов в поддержку творчества обратить внимание на другие области деятельности, как началась моя собственная одиссея. Я стал замечать, что такое значение дисциплине, формализму и количественному подходу придается не зря. Как бы ни нуждалась какая-то сфера деятельности в творчестве,

всюду необходима более формальная основа для его поддержки и укрепления.

Эта мысль часто проскальзывала в довольно неожиданных местах. В книге, многозначительно озаглавленной «The Discipline of Curiosity» (Дисциплина и любознательность) [Groen 1990]), автор писал: «Наука – не просто любознательность и творчество, это их дисциплинированная форма... Именно странный на первый взгляд союз дисциплины и любознательности обеспечивает научный прогресс. Любознательность творческого ума, заставляющая непрерывно задавать вопросы „как?“ и „почему?“, и дисциплина, чтобы понять: наука – это часть мира, и она влияет на него».

В другой книге, посвященной решению задач [Judson 1980], автор передает свой разговор с Мюрреем Гелл-Манном, открывателем кварков: «Со всяким искусством, если оно заслуживает этого названия, связана некоторая дисциплина. Даже если речь идет не о сонетах, симфонической музыке или классической живописи, а о весьма раскрепощенном современном искусстве, то и там действуют те или иные законы. И задача в том, чтобы донести то, что вы хотите, соблюдая эти законы».

И еще, из интервью с виолончелистом Ма Йо-Йо [Shapiro 1991]: «Творчество... не инстинктивно. Скорее это непрерывная борьба дисциплины с интуицией. Дисциплина – годы упражнений, музыкальное образование – основа исполнительского мастерства. Но в какой-то момент музыкант должен довериться своим чувствам...»

Одна и та же мысль повторялась снова и снова. В науке требуется как дисциплина, так и творчество. В изобразительном искусстве требуется как дисциплина, так и творчество. В музыке требуется как дисциплина, так и творчество. Очевидно, к моему разочарованию, и в программировании также требуются и дисциплина, и творчество!

Моя одиссея была почти завершена. Теперь я понимал, почему там, где творчество представлялось столь важным для достижения успеха, постоянно вставал вопрос дисциплины. Но тогда возникла новая дилемма: есть ли вообще смысл писать книгу?

Я перечитал компьютерную литературу, вновь отыскивая те места, где говорилось о дисциплине, формализме и количественном подходе. И снова, несмотря на мою одиссею, во мне поднялось возмущение. Я увидел, что защитники дисциплины и формализма делали ту же ошибку, что и я. Они полагали, что формализм продуктивен, а творчество нет. Совсем, как я, когда считал необходимым условием только творчество, а формализм – помехой.

Возможно, книгу все же стоило писать! Она могла бы оказаться полезной тем, кто слишком далеко отклонялся вправо, подобно тому как я откло-

нялся влево, помогая им увидеть, что правильный подход к решению задач расположен где-то между этими крайностями. А тем, кто не был столь предвзят, как мои «оппоненты» и я сам, она могла бы помочь разобраться в сути весьма важной проблемы, существующей в нашей области.

Мне не понадобилось менять собственные коренные убеждения. Тем не менее пришлось расширить их, включив в них этот «странный союз», эту «бесконечную борьбу», взаимодействие между одним существенным компонентом, творчеством, и другим существенным компонентом, дисциплиной. Моя одиссея наконец-то завершилась.

Мое пожелание и моя надежда – чтобы, читая эту книгу, вы проделали то же путешествие, которое совершил я, и остались довольны тем, что отправились в него.

Роберт Л. Гласс
Лето 1994 г.

Ссылки

GROEN 1990 – *The Discipline of Curiosity*, Elsevier Science, 1990; Groen, Jenny; Eefke Smit; Juurd Eijsvoogel.

JUDSON 1980 – *The Search for Solutions*, Holt, Reinhart and Winston, 1980; Judson Horace Freeland.

SHAPIRO 1991 – «Yo-Yo and Manny», *World Monitor*, Aug., 1991; Shapiro Michael.

Предисловие ко второму изданию

Забавно получилось с этой книгой.

Первое издание поначалу оказалось коммерчески неудачным. Книга плохо продавалась, авторский гонорар был невелик, отклика читающей публики почти не было.

Я был огорчен, но не пал духом. Чтобы написать эту книгу, я проделал большую исследовательскую работу, и то, что я узнал в результате, имело для меня большое значение. Я доволен тем, что, работая над книгой, сильно вырос интеллектуально. Из нескольких написанных мной книг – на сегодня – этой я горжусь, пожалуй, больше всего. В ней достигнут правильный баланс – по крайней мере, по моим представлениям – легко воспринимаемого стиля изложения и должного уровня научных исследований.

Спросите, что тут забавного? Дело в том, что, несмотря на слабое начало, количество проданных экземпляров стало расти. На конференциях и по электронной почте люди сообщали мне, как им понравилась книга. Все чаще и чаще. У меня появилось ощущение, что книга становится культовой. На рубеже тысячелетий я понял, что в мире программирования ее начинают воспринимать во многом так же, как я сам.

Поклонников книги становилось все больше. Остававшиеся в издательстве экземпляры были распроданы. Теперь, когда книги уже не было ни в печати, ни на складе, подорожали подержанные экземпляры. Из жадности я продал на *Amazon.com* свой запасной экземпляр почти за 100 долларов! Цена продолжала резко подниматься. Прямо сейчас, сочиняя новое предисловие, я открыл окно Amazon и смотрю на цену – 1196 долларов! Трудно поверить, что написанное мной может столько стоить! Жизнь явно требовала новой и улучшенной версии 2.0 «Креативного программирования». Работа началась.

Я привел имевшиеся заметки в соответствие с сегодняшним днем (за 10 лет в мире компьютеров и программирования многое произошло!). Я добавил несколько новых статей, чтобы отразить изменения и новые веяния.

Я постарался отразить гибкую методологию разработки (Agile), поскольку она близко связана с понятиями гибкости, творчества и качественного подхода, с которых началась моя одиссея (спешу добавить, что им может понадобиться пройти отчасти тот же путь, что и мне, чтобы приспособиться к представлениям о дисциплине и формализме, ставшим результатом моей одиссеи).

Я также добавил материал о методике Open Source – возможно, одном из последних бастионов представления об удовольствии как о мотивирующем факторе создания программного обеспечения.

Однако сердцевину книги обновлять не потребовалось. Одиссея, описанная мной в предисловии к первому изданию, все так же актуальна.

Замечательно, что все так забавно получилось. Кстати, благодарю всех поклонников книги за то, что исчезнувшая, не издаваемая и отсутствующая на прилавках книга стала пользоваться таким высоким спросом, который превзошел мои самые смелые мечты!

А тем новым читателям, которые впервые встречаются с «Креативным программированием», желаю радости от прочтения!

Роберт Л. Гласс
Лето 2006 г.

Почему «Креативное программирование»?

Для творческого мышления нужна позиция, сочетающая поиск идей и умелое обращение со своими знаниями и опытом. Встав на эту точку зрения, вы опробуете разные подходы,... часто не приводящие никуда. Вы используете безумные, дурацкие и непрактичные идеи, чтобы достичь практичных новых идей. Время от времени вы нарушаете правила и ищете новые идеи в непривычных местах.

Рождер фон Эйк

«A Whack on the Side of the Head»,
Warner Books, 1990

Миру программирования присущ принципиальный конфликт, иногда принимающий характер войны.

С одной стороны – руководители, которые стремятся навязать разработчикам более высокий уровень дисциплины и контроля, и исследователи, которые с теми же целями предлагают, а порой даже пытаются узаконить формальные методы.

С другой стороны – разработчики программного обеспечения, спокойно продолжающие писать программы обычным путем, то есть свободно выбирая методы и применяя творческие решения.

Время от времени появляются «революционные» методы, которые какое-то время пользуются популярностью и затем исчезают, мало что изменив в существующей практике разработки программ.

Методологии дробятся, языки программирования приходят и уходят, CASE-средства пылятся на полках, ориентация на объектность вечно остается «задачей следующего года». Попытки упростить и регламентировать разработку программного обеспечения в целом проваливаются одна за другой.

В чем причина такой ситуации? Может быть, в неподатливости разработчиков, которые настолько закоснели в своих привычках, что не способны воспринимать новое? Похоже, руководители и исследователи именно так и полагают.

А может, дело в невежестве руководителей и исследователей, которые настолько озабочены поиском новых решений, что перестали понимать старые? Такая точка зрения характерна для практиков.

Думаю, главная проблема этого конфликта – творчество. Если при создании программ творчество почти или совсем не нужно, то правы руководители и исследователи, заявляющие, что процесс изготовления программ можно упростить и регламентировать. Если же, напротив, творчество было и остается важной составной частью процесса создания программ, то правы практики: нам по-прежнему будут нужны свободные методы и творческие решения. Вот эту-то совокупность проблем, возникающих из представлений о творчестве в программировании, я и собираюсь рассмотреть в книге и, если удастся, решить. Потому я и назвал ее «Креативное программирование».

Рассмотрению этих диаметрально противоположных точек зрения и возможных путей выхода из конфликта посвящена первая часть книги. Я разделю проблему на составные части. Вместо того чтобы рассматривать творчество в целом, мы последовательно изучим его с разных сторон, взяв такие его аспекты, как дисциплина, формальные методы, оптимизация решений, количественный подход, технологический процесс, теория и занимательность. Эти и некоторые другие темы будут изучены в ряде неформальных очерков. Попутно мы рассмотрим пару дефиниций творчества и даже результаты исследования, имевшего целью выяснить, какие же задачи разработки программного обеспечения действительно требуют творческого подхода.

Затем в части II мы перейдем к несколько более формальной и практической точке зрения на творчество. Мы возьмем две области, в которых творчество играет особо важную роль – творчество в организационной сфере и творчество в технологиях программирования, – и изучим значение творчества в них существенно глубже. А после, чтобы более полно

осветить творчество в программировании, мы взглянем на наш предмет с исторической точки зрения: где и как возникла проблема творчества? Для каждой из этих тем мы рассмотрим пути стимуляции творчества. Для этого есть чертова дюжина приемов, которые мы опишем и обсудим.

Чтобы не ограничиваться при рассмотрении творчества точкой зрения одних лишь программистов, в части III мы рассмотрим то, что узнали о творчестве, применительно к другим областям. В частности, нас ждет несколько удивительных ответов на вопрос: «Могут ли компьютеры и программы повысить уровень творчества в других (некомпьютерных) областях?».

В части IV мы увяжем свободные концы всех нитей книги.

Это обзор структуры книги, но как она построена с эмоциональной точки зрения? Куда она приведет нас в итоге?

Надеюсь, по мере чтения книги вы станете по-новому воспринимать роль творчества и тот конфликт, с рассказа о котором я начал. Где-то я подам идею, где-то подскажу – все это поможет вам сформировать собственное мнение относительно важности творчества в разработке программного обеспечения. Изучение материалов для этой книги стало для меня путешествием чувств и ума, уведя далеко от того места, где все началось. Такое может случиться и с вами.

Попутно я призываю на помощь некоторых коллег. Например, у известного программиста П. Дж. Плогера (P. J. Plauger) мы позаимствуем понятие «индекса сложности» («the falutin' index») – легкомысленного способа оценки программных проектов, весьма отличающегося от формальных метрик, о которых можно прочесть в формальной литературе. Аналогично Брюс Блюм (Bruce Blum) из Лаборатории прикладной физики имени Джона Хопкинса поможет нам изучить соотношение между развлечением и скукой в программных проектах, а Дуэйн Филипс (Dwayne Phillips) будет руководить нашим исследованием забавной стороны некоторого проекта. Деннис Галетта (Dennis Galletta), профессор из Питта, расскажет о творчестве в стратегических системах. Покойный Дэн Кугер (Dan Couger) из Университета Колорадо (и директор Центра изучения творчества в Колорадо-Спрингс) – автор ряда идей о творчестве в программировании, которые я позаимствовал из его книг. С Айрис Весси (Iris Vessey) из Университета Пенсильвании мы вместе исследовали природу задач программирования и описали результаты. Таким образом, данная книга отчасти является коллективным трудом.

Часть I

Два типа мышления: исследование творчества в области программирования

Введение
1. Дисциплина и гибкость
2. Формальные методы и эвристики
3. Оптимизация и разумная достаточность
4. Количественный и качественный подходы
5. Процесс и продукт
6. Интеллектуальные и канцелярские задачи
7. Теория и практика
8. Наука и производство
9. Забавность и серьезность

Введение

Наиболее успешно решают задачи те, кто обладает редким умением объединять режимы сознательного функционирования, свойственные разным полушариям мозга, переключаясь в обоих направлениях между целостным и последовательностным, между интуицией и логикой, между расплывчатым пространством проблемной области и маленьким четким конкретным участком этого пространства. Это могут быть выдающиеся художники и ученые, потому что они сочетают в себе сильные стороны как... Леонардо да Винчи ... так и Эйнштейна.

Моше Ф. Рубинштейн

«Tools for Thinking and Problem-Solving»,
Prentice-Hall, 1986

К написанию книги о творчестве в программировании можно подойти по-разному:

- слегка коснувшись проблем творчества, привести массу советов по поводу того, как проявлять больше творчества, и множество примеров их успешного применения;
- достаточно серьезно и академически, переработав массу литературы, дословно описать все, что известно по различным аспектам данной темы;
- изложить неофициальный взгляд на важность творчества для разработки программного обеспечения.

В этой книге нашли отражение все три подхода, хотя автор явно склоняется к последнему. В части I ему отдано предпочтение. Следующие статьи в целом представляют собой неофициальное собрание мыслей о важности творчества в программировании.

Как мне кажется, данная проблема описывается рядом известных и считающихся ключевыми проблем в области программирования:

- роль дисциплины;
- роль формальных методов;
- роль оптимизации решений;
- роль количественного подхода;
- роль технологического процесса;
- роль теории;

- роль науки;
- роль занимательности.

Думаю, в действительности каждая из перечисленных проблем имеет отношение к творчеству. Уверен, что при творческом подходе к созданию программ мы:

- освобождаемся (хотя бы частично) от дисциплины;
- переходим от более формальных методов к менее формальным;
- выбираем решения, достигающие цели (возможно, не самым оптимальным образом);
- отдаем предпочтение не количественному, а качественному подходу;
- интересуемся как процессом, так и его результатом;
- понимаем, что иногда успешная практика опережает успешную теорию и что некоторые секреты производства науке неизвестны;
- и получаем больше удовольствия от работы!

В духе этой дихотомии – что творчество означает разрыв со стандартными шаблонами – я и выбрал слова «типы мышления» для названия этой части книги. Каждый из следующих заголовков означает противопоставление: формальных методов – эвристикам, количественного подхода – качественному, практики – теории. Уверен (и книга подтверждение тому), что в современном мире дисциплинированному, формальному подходу к разработке программного обеспечения придается чрезмерное значение. Я хочу сказать, что есть не только альтернативы общепринятому здравому смыслу, но и веские основания перейти от него к более взвешенной точке зрения на рассматриваемую область деятельности.

В этой части не только даются ответы, но и ставятся новые вопросы. Всегда ли дисциплина – правильный метод для программного проекта? Права ли компьютерная наука, делая столь сильный упор на формальные методы? Верно ли, что «нельзя управлять тем, чего не можешь измерить»? Всегда ли теория опережает практику? Должны ли серьезные размышления вытеснить элемент развлекательности в нашем деле? Пытаясь ответить на эти вопросы, я сталкиваюсь с другими и ставлю новые.

Однако есть и однозначные ответы. Научное исследование задач программирования позволило выяснить, в какой мере эти задачи являются интеллектуальными, канцелярскими и творческими. Некоторые из результатов этого исследования могут удивить вас.

Глава 1 | Дисциплина и гибкость

Введение
1.1. Генри Форд от программирования, пожалуйста, встаньте!
1.2. Автоматизация программирования – факт или фикция?
1.3. Правда ли, что программистами «невозможно управлять»?
1.4. Дисциплина – неприличное слово: рассказ о жизненном цикле разработки программного продукта
1.5. Имитация программной разработки
1.6. Гибкое программирование (Agile): гибкость достигла зрелости
1.7. Странный случай с карандашом корректора
1.8. Индекс сложности
1.9. «Странная парочка» – дисциплина и творчество

Введение

Наука – не просто любознательность и творчество, это их дисциплинированная форма... Именно странный на первый взгляд союз дисциплины и любознательности обеспечивает научный прогресс. Любознательность творческого ума, заставляющая непрерывно задавать вопросы «как?» и «почему», и дисциплина, чтобы понять, что наука – это часть мира, и она оказывает на него влияние.

Дженни Гроэн

«The Discipline of Curiosity», Elsevier Science, 1990;
Janny Groen, Eefke Smit, Juurd Eijsvoogel

Работая над этой книгой, я с трудом подобрал понятие, противоположное «дисциплине». Очевидный кандидат «недисциплинированность» был отвергнут из-за негативного оттенка. При том, что никто и не собирается предлагать полный отказ от дисциплины при разработке программного обеспечения. Всякий раз, возвращаясь к этой теме в надежде на озарение, я вновь оказываюсь у пустого колодца. Я бы хотел различать управляемое поведение в едином русле (дисциплинированное) и целенаправленное, но не связанное ограничениями поведение (творческое).

Я не придумал ничего лучше, чем «гибкость». Имея в виду такие аспекты дисциплины, как подчинение и контроль, я выбрал «гибкость» в противоположность им. В моем словаре «гибкий» (flexible) означает «способный легко гнуться, не ломаясь; приспосабливающийся, способный меняться в зависимости от обстоятельств».

Меня устраивает это определение, хотя я не вполне уверен, что правильно противопоставил его дисциплине. То есть этот вариант меня не совсем удовлетворяет, но, думаю, сгодится и он. Если найдете более удачное слово, мысленно подставьте его в заголовок этой главы!

Закончим на этом игры со словами. Почему в книге о творчестве в программировании появилась глава о дисциплине и (...э-э...) гибкости? Потому что одно из древнейших противостояний в нашей профессии связано именно с этим вопросом: какой должна быть команда разработчиков программного обеспечения – дисциплинированной, как на заводе, или гибкой и с высокой личной мотивацией?

Разумеется, это противостояние старше программирования. Когда мир перешел от кустарного производства к фабричному, стало совершенно яс-

но – по крайней мере, на тот момент, – что будущее принадлежит дисциплинированным, а не гибким способам организации. Век автомобилей, конечно, не настал бы без Генри Форда с его идеей неквалифицированной, но дисциплинированной рабочей силы.

Но это было давно, а сейчас времена переменились. Многие согласятся, что эру фабричных труб сменила эра информации. Общество, которое раньше зависело от умения что-то производить из дорогого сырья на огромных фабриках, теперь зависит от информации, обрабатываемой с помощью дешевого кремния. Будущее принадлежит работникам сферы знаний, а не заводским рабочим. Вопрос в том, как найти оптимальные способы организации и управления командами работников сферы знаний.

Несомненно, разработка программного обеспечения – это вершина обработки информации. Если вы согласны хотя бы отчасти поверить заявлениям Дэвида Парнаса и Фреда Брукса, что «программировать тяжело» и что «это самый сложный вид деятельности, которым когда-либо занимались люди», то нетрудно предположить, что прежние модели организации труда на фабрике оказываются непригодными в эпоху информации.

Любопытно отметить, что сегодня успешные компании-производители программного обеспечения финансируют разработку программ по бюджетной статье «Исследования и разработки», то есть рассматривают создание программ не как вид промышленного производства, а как деятельность, более близкую к интеллектуальному труду исследователя.

А теперь задумаемся об исследованиях и разработках. Характерны ли для них «подчинение» и «контролируемое поведение»? Разумеется, нет. Дисциплина может иметь значение для исследований и разработок (мы глубже изучим эту тему в последующих разделах), но гораздо важнее здесь гибкость – или другое слово, которое вы подберете для этого понятия.

В таком случае соотношение ролей дисциплины и гибкости должно задаваться моделью наших представлений о разработке программного обеспечения. Что правильнее: модель фабричного типа, когда рабочие целый день штампуют детали и собирают из них программу на конвейере, или что-то вроде мастерской или лаборатории, когда разработчики целый день решают интеллектуальные проблемы?

Этой проблемой мы и займемся в данном разделе. Вопрос не нов: чтобы показать это, я включил в книгу (на нашем профессиональном языке можно сказать «повторно использовал»!) статью «Генри Форд от программирования, пожалуйста, встаньте!» примерно тридцатилетней давности, но, как мне кажется, хорошо описывающую проблему. А заключительный раздел «„Странная парочка“ – дисциплина и творчество» отражает мое личное разрешение данного противоречия.

Вперед! И заодно подумайте о том, как бы вы сами разрешили противоречие между дисциплиной и гибкостью.

1.1. Генри Форд от программирования, пожалуйста, встаньте!

Расстояние между штаб-квартирами фирм Alchemy Chemical и Softli Paper не больше мили, если ехать по Прямому шоссе в Кремниевом содружестве (Массачусетс).

Но их производственные мощности с таким же успехом могли располагаться на Марсе и Венере, потому что их организационную философию разделяют световые годы. И занимаемые ими крайние позиции характерны для современных дилемм в программировании.

В Alchemy твердой рукой правит Стэн Сорсер. Его служащие придерживаются установленного дресс-кода, работают от звонка до звонка, пьют кофе в отведенные для этого перерывы и строго выполняют правила поведения. Для программистов установлены многочисленные и обязательные стандарты. Документация и листинги подвергаются периодическому контролю и внезапным проверкам. Все приложения пишутся только на C# и Java, а для добровольно принудительного соблюдения стандартов программирования применяются средства статистического анализа. Руководство строго следит за выполнением графика работ, и если программист не укладывается в срок, он должен ликвидировать отставание, добровольно работая сверхурочно. Обстановка в Alchemy напряженная, деловая и продуктивная. Стэн любит демонстрировать данные, согласно которым производительность труда у его программистов втрое выше, чем в среднем по стране. По всем внешним признакам Стэн построил фабрику, производящую программное обеспечение, и руководит ей, как сборочным конвейером.

А глава Softli Paper Херб Бонд смотрит на все иначе. Его программисты одеваются в стиле «неохиппи». Двери Softli открыты круглые сутки, программисты приходят и уходят. Инструкция по стандартам программирования здесь тоньше предметного указателя Alchemy и вместо «должен» и «обязан» в ней говорится «может» и изредка «следует». У Херба принято, чтобы код рецензировали коллеги-программисты, а не проверяли начальники. Из языков здесь преобладают C# и Java, но один из асов-программистов пишет сейчас генератор отчетов на Python и очень доволен. Управление заданиями осуществляется с помощью отчетов о ходе работы, причем качеству продукта уделяется больше внимания, чем соблюдению графика. Свободная, неказенная атмосфера в Softli напоминает семинары на старших курсах. Херб никогда не приводит статистические

показатели по продуктивности, но как-то обмолвился, что не доверяет им, зная, откуда берутся эти фиктивные числа. По всей видимости, Херб Бонд пасет табун диких необъезженных программистов, направляя его движение в сторону выпуска готового продукта.

Совершенно очевидно, что Стэн Сорсер и Херб Бонд не могут быть оба правы относительно того, как надо управлять программным производством. Программисты должны работать либо как рабочие на конвейере, либо как художники. И все же – как?

Ответить на этот вопрос не так легко. Десять лет назад казалось, что философия сборочного конвейера – единственно верная. Программные продукты отличались чрезмерной сложностью, непомерной дороговизной и отставанием от графика. Программные модули изготавливались и собирались вручную мастерами. Казалось очевидным, что характерные для программных продуктов проблемы обусловлены кустарным способом производства.

Исходя из такого предположения руководство выделило немалые средства на разработку технологий конвейерной сборки программных продуктов. Первые надежды возлагались на Visual Basic – язык, поддерживающий перетаскивание и мастера для разработчика, с помощью которого даже любители и начинающие программисты могли создавать приложения с невиданной ранее скоростью.

Однако расчеты не оправдались. Как ни крути, программы остаются программами. Их индивидуальные особенности и сложность таковы, что одним перетаскиванием не обойтись. Visual Basic не смог изменить ситуацию (как позже не смогли Java и C#).

Но руководство не теряло надежд. Поощрялись технологии, предлагавшие стандартизацию. Значение индивидуальности принижалось, превозносилась групповая работа. Возник термин «неэгоистичное программирование». Были формализованы управление настройками и контроль качества. А когда появилось автоматизированное управление жизненным циклом, сулившее чудеса, руководство с готовностью ухватилось за него. Казалось, что земля обетованная – поточная линия программирования – уже на горизонте.

Но в этом стремлении к максимальной автоматизации была изрядная доля иронии. В то время как исследователи отдавали все свои силы разработке технологий конвейера, почти никто не уделял внимания мастерству технологов. Несмотря на запуск конвейеров программные продукты по-прежнему создавались мастерами-умельцами. И никто не занимался разработкой и продвижением усовершенствованных инструментов для этих умельцев. Ну, то есть не совсем так. Были технологи, которые создавали более эффективные инструменты. Но почти никто не обращал на

них внимания. Если новая технология не предвещала быстрого приближения к поточной линии, редкий руководитель удосуживался вникнуть в нее. А без поддержки руководителей новые инструменты не могли попасть к программистам-практикам.

Хочу внести полную ясность в один вопрос. Я не отрицаю, что в один прекрасный день программирование может встать на поток. Не исключено, что где-то уже сейчас некий Генри Форд от программирования разрабатывает совершенный метод изготовления программ. Но пока исследователи и инвесторы заняты этим делом, кто-то должен решать более близкие задачи. Глупо поспешно переводить отрасль на конвейерное производство, когда никто не знает, как проектировать этот конвейер. И глупо прекращать разрабатывать инструменты и применять их, оттого что когда-нибудь они, возможно, больше не понадобятся.

Однако хватит разглагольствовать. Я, пожалуй, слишком отклонился от Прямого шоссе, Стэна Сорсера и Херба Бонда.

А может быть, и нет. Фирма Стэна Сорсера – это высшее современное достижение в организации конвейера. А фирма Херба Бонда – вершина подхода, основанного на привлечении творцов. И мои разглагольствования были попыткой объяснить, почему в действительности нельзя утверждать, что Стэн с его цифрами прав, а Херб – нет. Или наоборот.

Но если вернуться к Alchemy и Softli, то очевидно, что противоречие между этими двумя компаниями не прошло незамеченным для их сотрудников. Дело в том, что в пятницу вечером в баре на Прямом шоссе бывало очень интересно, если там сходились люди из Alchemy и Softli. В один из таких пятничных вечеров и произошло описываемое ниже событие.

Представьте картину. Ровно в 16:43 у бара останавливается пара машин с программистами Alchemy, возвращающимися с работы. Одетые строго по дресс-коду в рубашки поло и модельные джинсы, они с некоторым презрением поглядывают на бородатых парней из Softli в мешковатых шортах и сандалиях, потягивающих в уголке свои напитки. Взгляд не остается незамеченным, и один из двоих программистов Softli, засидевшийся в баре дольше, чем позволяет истекающий завтра срок сдачи работы, презрительно отзывается о стиле одежды и родословной программистов Alchemy в целом и данной их группы в частности.

Если бы в баре собирались заводские ребята, уже мелькали бы кулаки. Но заведению присуща некоторая интеллектуальная атмосфера, и вместо этого возникла изящная перебранка. В ход пошли разные идиоматические выражения. Сражение еще только началось, а дресс-код Alchemy уже был назван *гитлеровским*, манера одеваться в Softli – *стилем бездельников*; распорядок дня Alchemy – *бухгалтерским*, продуктивность Softli – *высо-*

кой, но отрицательной, дисциплинированное программирование в Alchemy – *упорядоченным кретинизмом*, а характер труда в Softli – *детсадовским*. Перепалка разгоралась.

Не знаю, кто первым предложил провести турнир. И не уверен, что в конечном счете это была верная идея. Но остановить сражение удалось. Высказанная идея прозвучала как колокол в суде, заставив всех умолкнуть, после чего началось бурное обсуждение условий. Идея была крайне проста, типа «мой папа побьет твоего папу». «Сделай, что можешь, или заткнись!» – вот и весь разговор. «Если вы такие способные программисты, посмотрим, сможете ли вы нас переплюнуть».

Идею подхватили все. Когда вечером эта разношерстная публика покинула бар, основные правила были уже оговорены.

Турнир был достаточно прост. Команды от обеих компаний должны были написать две программы. Первая – генератор отчетов, относительно эффективный и незамысловатый. Вторая – небольшой интерпретатор языка с парой сложных типов данных и мудреной семантикой. Оцениваются время написания программы, стоимость написания и качество готового продукта. Качество, естественно, определяется субъективно с учетом некоторой таблицы атрибутов качества, призванной способствовать объективности оценки и составленной судейской бригадой из сотрудников обеих компаний. Проводить соревнования планировалось по утрам в баре в течение четырех суббот подряд.

В эти субботы в баре не смотрели кубковые футбольные матчи. Помещение было набито болельщиками из обеих компаний, потому что весть о турнире разнеслась по всем ИТ-отделам. Перфораторщицы из Alchemy в одинаковых свитерах и брюках подобающей скромной расцветки смотрелись как импровизированная группа поддержки. Группа штатных аналитиков из Softli организовала блог и RSS-трансляцию, чтобы сообщать о ходе соревнований в реальном времени. На третью субботу пришло местное отделение Ассоциации по вычислительной технике в полном составе, а на четвертую явились репортеры из «Прямошоссейного вестника» и «Трубного гласа». Событие стало крупнейшим в Кремниевом содружестве после урагана Гизельда.

Результаты турнира – которых вы ждали, затаив дыхание, – разочаровали. Если не придираться к подсчету очков, то получилась ничья. Конвейерщики из Alchemy легко справились с генератором отчетов и показали хорошие результаты по времени и стоимости. Зато умельцам из Softli лучше дался интерпретатор, и баллы за качество у них были выше. Для такого рода соревнований исход оказался, видимо, лучшим из всех возможных – проигравших не оказалось и никто не испытывал неловкости, когда в последнюю субботу толпа расходилась из бара.

Хотя иногда я задаюсь вопросом, что бы произошло, победи одна из команд. Например, если бы оказалось, что конвейер действительно эффективнее мастеров-умельцев. Или наоборот.

Можете ли вы представить, что Стэн Сорсер оказался неправ и ему пришлось разрешить своим программистам приходить на работу в сандалиях? Или что Херб Бонд убедился, что ошибался, и потребовал от своих программистов работать с восьми до половины пятого? Во всяком случае я рад, что Генри Форд от программирования пока не объявился. Хотя не исключено, что в будущем году мы увидим его в этом баре – как раз на втором ежегодном турнире программистов Силиконового содружества!

1.2. Автоматизация программирования – факт или фикция?

«Производство программ без суеты и спешки». Вот соблазнительная картина, обрисованная в [Moad 1990] и тысячах других статей, и по сей день появляющихся в компьютерной прессе, которые расписывают, как станут разрабатываться программы в будущем. Но реально ли это?

Замах известен. «Мы создали более 4000 новых программ, к которым не было претензий, и сократили сопровождение на 70–90%», – заявляет возбужденный руководитель, применивший некую новую технологию. «Мы можем выпускать [приложения] с такой скоростью, с какой они [пользователи] способны осваивать их», – продолжает он свои излияния [Moad 1990].

Однако есть и оборотная сторона. Многие разработчики не испытывают энтузиазма по поводу тех же самых новых технологий. Они просто упрямы – или блестящие отчеты об успехах не во всем убедительны? Сможет ли передовая технология – некоторые называют ее автоматизацией программирования – оправдать возлагаемые на нее надежды?

Наряду с бодрыми сообщениями поступают и трезвые предупреждения: «Многие сохраняют чувство настороженности, поскольку в прошлом мы часто терпели неудачи», – говорит другой руководитель. «В настоящее время редко кто согласится публично признать, что инструменты анализа и проектирования высокого уровня неэффективны. Кто же захочет стать недобрым вестником?» – вторит ему еще один.

Несомненно, на карту поставлено многое. Общеизвестно, что повсюду в мире разработка программ обходится дорого. Кто не знает, что за последние десятилетия эффективность программирования выросла весьма незначительно? Конечно, все мы, кто занимается программированием, хотим, чтобы эти показатели каким-то чудесным образом изменились в лучшую

сторону. Но достаточно ли для этого одних пожеланий – вместе с определенными усилиями?

Некоторые считают, что да. Обычно это продавцы решений и те, кто хочет получить грант на поиск новых решений. Недостатка в обещаниях прорыва в технологиях создания программ не наблюдается.

Некоторые считают, что нет. Обычно это те, кто долго зарабатывал себе на жизнь программированием и повидал немало обещаний. Недостатка в не верящих обещаниям прорыва тоже нет.

Линия фронта определена. Но кто прав?

К сожалению, нет данных в пользу той или иной позиции. Разумеется, есть много удивительных отдельных случаев. Тот описанный выше человек, который создавал программы с такой скоростью, что пользователи не успевали их осваивать, – это один пример. Бывают еще руководители, которые обучают всех своих подчиненных какой-нибудь новой «революционной» технологии, потом забывают об этом и через несколько лет пытаются внедрить конкурирующую и несовместимую методологию, потому что она-то уж точно вызовет революцию в программировании. Примерами можно обеспечить любую точку зрения.

Однако медленно, но верно растет число тех, кто сомневается в претензиях подобного рода. Одним из первых по этому поводу высказался известный в компьютерной науке специалист Дэвид Л. Парнас (David L. Parnas), изложивший свои сомнения в статье «Software Aspects of Strategic Defense Systems» (Программные аспекты стратегических оборонительных систем) [Parnas 1985]). Затем известный гуру программирования Фред Брукс (Fred Brooks) выразил аналогичные и многие другие свои сомнения [Brooks 1987]. Два специалиста по исследованиям в области автоматизации программирования заявили о своих сомнениях по поводу чрезмерно завышенных обещаний (и даже назвали их мифами!) [Rich 1988].

Хор сомневающихся набирал силу. Карма Маклюр (Carma McClure), бывшая когда-то одним из самых ярых энтузиастов автоматизации, теперь отзывается о движении CASE как о «голом короле», заявляя, что применение CASE практически не сказалось на росте производительности [SD 1990]. Мое собственное обширное исследование подобных «революционных» концепций [Glass 1999] показывает, что фактов, подкрепляющих какие-либо из подобных претензий, нет.

Возможно ли, чтобы эти (допустим, весьма личные) мнения были верными? Правда ли, что никакой реальный прорыв здесь невозможен? И что единственная реалия – существование людей, верящих в возможность таких прорывов?

Здесь необходимо прибегнуть к старому доброму научному методу исследований. Выдвигается гипотеза о том, что некий конкретный метод способен «революционным» образом повысить эффективность программирования. Предлагается схема исследования, которое должно выяснить справедливость или ложность этой гипотезы. Это может быть эксперимент, опрос или анализ регистрационных данных, полученных от тех, кто активно применяет новую идею и ее прежние альтернативы. Есть много правильных способов организации исследований. Не хватает не умения проводить такие исследования, а желания и, к несчастью, финансирования (такие исследования недешевы).

Короче, следует и перестать пропагандировать закосневшие точки зрения, свойственные нашим представлениям о состоянии «автоматизации программирования» в 2006 году (и, кстати, в 1996, 1986, 1976 и 1966 годах!), и постараться объективно взглянуть на ситуацию, проведя исследование с намерением выяснить истину.

К счастью, примеры такого рода работ есть. Многое в этом направлении делается в лаборатории программирования центра НАСА в Годдарде, о чем в течение многих лет сообщается на ее ежегодных семинарах. Кое-что достигнуто сообществом, разрабатывающим теоретические основы работы программистов, – участниками конференции «Empirical Studies of Programmers», часть которых стала издавать уважаемый научный журнал на тему эмпирических исследований в программировании. Отдельные работы ведутся в различных исследовательских институтах, таких как институт Фраунхофера в США и Германии, NICTA в Австралии и Simula в Норвегии.

Есть также несколько хороших работ отдельных исследователей, которых интересует истина, а не пропаганда каких-то идей. Маленький пример – работа [Davis 1988], содержащая исключительно вдумчивый и конкретный анализ того, чем хороши CASE-средства и чем они плохи.

Если к таким исследованиям подключится достаточное число людей как в науке, так и в промышленности, и в таких учреждениях, как Институт программирования (Software Engineering Institute), не исключено, что мы сможем лучше разобраться в основной проблеме, которая здесь кроется.

Некоторые утверждают, что разработка программ – наука, или должна стать ею. Все нужно строго и теоретически обосновать. Для облегчения работы добавить немного математики. И есть ощущение, что при научном подходе и достаточной дисциплинированности можно автоматизировать программирование.

Другие говорят, что это скорее искусство. Как можно больше творчества и раскрепощенной мысли. Для облегчения такой работы добавить немно-

го технического мастерства. Есть ощущение, что дисциплина и автоматизация просто не эффективны.

Один руководитель, которого цитирует [Moad 1990], выразил это лучше всех: «Мы пытаемся приблизить разработку программ к науке, но выяснилось, что среди людей есть ученые, а есть художники. Многие из наиболее творческих наших работников являются художниками, и мы не хотим потерять таких людей».

Должным образом проведенные исследования могли бы показать, как лучше всего использовать ученых, а как – художников. Как и в большинстве вопросов, когда есть укоренившиеся противоположные взгляды, истина, скорее всего, находится где-то между ними.

Ссылки

BROOKS 1987 – «No Silver Bullet», *IEEE Computer,* April 1987; Fred Brooks.

DAVIS 1988 – «Commentary», *Accounting Horizons,* June 1988; Gordon B. Davis.

GLASS 1999 – «The Realities of Software Technology Payoffs», *Communications of the ACM*, Feb., 1999; Robert L. Glass.

MOAD 1990 – «The Software Revolution», *Datamation,* February 15, 1990; Jeff Moad.

PARNAS 1985 – «Software Aspects of Strategic Defense Systems», *American Scientist,* September 1985; David Parnas.

RICH 1988 – «Automatic Programming: Myths and Prospects», *IEEE Computer,* August 1988; Charles Rich and Richard C. Waters.

SD 1990 – «Report on the System Development Conference», *System Development,* April 1990.

1.3. Правда ли, что программистами «невозможно управлять»?

Известный компьютерный специалист часто вспоминал свое первое впечатление от работы в группе промышленной разработки программ: «Эти программисты неуправляемы».

Проблему контроля при разработке программного обеспечения совсем несложно представить. Типичный программист – нечесанная личность в драных джинсах и сандалетах, работающая во внеурочное время и поглощающая огромное количество вредной для здоровья пищи. И этот стереотип применим не только к программистам прошлых лет, но и к сего-

дняшним хакерам. Стоит ли удивляться, что руководитель, которому предстоит работать с людьми, похожими на анархистов, может опасаться проблем?

Проблема контроля, разумеется, тесно связана с проблемой дисциплины программистов. Контроль – это инструмент для установления дисциплины. Если дисциплина жизненно необходима для разработки программных продуктов, то контроль является важным средством в арсенале руководителя программного производства.

Несомненно, многим требуется больший контроль и большая дисциплина. Например, часто встречаются заявления вроде «...введение строгой дисциплины и механизации путем применения передовых методов программирования... дает нам практическое решение... Время для их введения настало...» [Lehman 1989].

Однако здесь есть коренная проблема. Способствует ли контроль повышению эффективности разработки программного обеспечения на самом деле? Существуют ли исследования, какие-либо данные, подтверждающие или опровергающие необходимость контроля? Правда ли, что программистами вообще невозможно управлять?

К сожалению, такие исследования раньше почти не проводились, да и сейчас их немного. Однако некоторые данные исследований по другим направлениям свидетельствуют о том, что контроль и дисциплина, возможно, не решают проблем разработки ПО, вопреки утверждениям некоторых.

Одно из таких открытий приведено у Демарко [DeMarco 1987]. Речь идет о результате более раннего исследования взаимосвязи между способами оценки сроков выполнения программных разработок и возможностью приблизиться к этим оценкам на практике [Jeffery 1983]. Были исследованы следующие ситуации:

1. Программный продукт разрабатывался по графику, установленному руководством.
2. Программный продукт разрабатывался по графику, установленному производственниками.
3. Программный продукт разрабатывался вообще без графика.

В исследовании измерялась продуктивность разработчиков. Результаты оказались отчасти предсказуемыми, отчасти совершенно неожиданными:

1. Продуктивность была выше, когда разработчики устанавливали собственный график, а не пытались действовать по графику, предложенному руководством. (Чего и следовало ожидать.)

2. Наивысшая продуктивность программирования достигалась при полном отсутствии графика. (Чего, конечно, никто не ожидал.)

Эти открытия можно интерпретировать по-разному. Трудно спорить с тем выводом, что будучи предоставлены самим себе (и, таким образом, оказавшись «вне контроля»), программисты демонстрируют свою высшую эффективность. Складывается впечатление, что программисты обладают внутренним чувством ответственности, которое в соединении с желанием хорошо делать свое дело мотивирует сильнее, чем любые попытки руководства оказывать давление и контролировать!

Есть аналогичные данные, полученные еще раньше. Леман в своей статье, посвященной методам управления, показывает, что чем жестче контролируют программистов, тем ниже их продуктивность [Lehman1979]!

Два свидетельства еще не гарантируют надежность выводов, но все же стоит задуматься над этими данными. С тех пор как программирование стало превращаться в профессию, ведется поиск наиболее эффективных способов управления программистами и приучения их к дисциплине. Предлагались следующие варианты:

- Управление по теории X: у работников отсутствует мотивация, и задача администрации – заставить их хорошо работать.
- Управление по теории Y: у работников высокая мотивация, и задача администрации – помочь им правильно выбрать задачу.
- Управление по теории Z: сочетание помощи и контроля – лучший способ добиться хорошей работы.

Считалось, что правильный метод управления программными разработками лежит где-то между теориями X и Z. Сделанные открытия указывают, что это не так и что искать нужно между теориями Y и Z.

Интересно, что в более традиционных областях науки лучше представляют нужные пропорции. Например, в двух приведенных ниже цитатах отмечается необходимость сочетать творчество и дисциплину, но не забудем, что дисциплина, о которой там говорится, должна быть в значительной мере *внутренней*, а не налагаемой извне:

> «Наука – не просто любознательность и творчество, это их дисциплинированная форма... самодисциплина... практического применения блестящих новых идей. Именно странный на первый взгляд союз дисциплины и любознательности обеспечивает научный прогресс. Любознательность творческого ума, заставляющая непрерывно задавать вопросы „как?“ и „почему?“, и дисциплина, чтобы понять: наука – это часть мира, и она влияет на него» [Groen 1990].

> «Со всяким искусством, если оно заслуживает этого названия, связана некоторая дисциплина. Даже если речь идет не о сонетах, симфонической музыке или классической живописи, а о весьма раскрепощенном современном искусстве, то и там действуют те или иные законы. И задача в том, чтобы донести то, что вы хотите, соблюдая эти законы» [Judson 1980].

Настало время разобраться в том, что понимается под большей дисциплиной и контролем в программировании. Если речь идет о самодисциплине, то с помощью контроля ее не добиться. А если об обычной дисциплине, устанавливаемой внешним контролем, то нужно понять, почему такая точка зрения плохо согласуется со странными результатами исследований.

Может удивить, что после 50 лет развития нашей области возникают такие фундаментальные вопросы. Но 50 лет – это очень небольшой срок в сравнении с другими областями знаний. Еще не поздно заново задать вопросы «как?» и «почему?» в администрировании программных разработок.

Ссылки

DeMarco 1987 – *Peopleware,* Dorset House, 1987; Tom DeMarco and Tim Lister.

Groen 1990 – *The Discipline of Curiosity,* Elsevier Science, 1990; Janny Groen, Eefke Smit, and Juurd Eisvoogel.

Jeffery 1985 – «Managing Programmer Productivity», *Journal of Systems and Software,* January 1985; Ross Jeffery and M.J. Lawrence

Judson 1980 – цитата из разговора с Мюрреем Гелл-Манном, открывшим кварки, в *The Search for Solutions,* Holt, Rinehart and Winston, 1980; Horace Freeland Judson.

Lehman 1989 – «Uncertainty in Computer Application and Its Control Through the Engineering of Software», *Journal of Software Maintenance,* September *1989; M. M. Lehman.*

Lehman 1979 – «How Software Projects Are Really Managed», *Datamation,* January 1979; M. M. Lehman.

1.4. Дисциплина – неприличное слово: рассказ о жизненном цикле разработки программного продукта

В прежние времена жизненный цикл программы был четким и ясным. Изучались требования задачи, которую нужно было решить. Создавался проект решения, удовлетворяющего этим требованиям. Согласно этому проекту писали код. Этот код тестировался. После этого программное решение выпускалось в свет, чтобы пользователи пользовались им, а сопровождающие – сопровождали.

Руководству так понравился этот жизненный цикл, что оно решило напрямую связать управление с ним. Буквально. Сначала, говорило оно, вы должны проанализировать требования. Потом – и только потом – можно начинать проектирование. И только когда готов проект, можно начинать писать код. И, разумеется, нельзя начинать тестирование, не завершив написание кода, а пользователи не могут получить продукт, пока не будут пройдены все тесты.

В неосязаемом мире разработки программ было очень приятно получить порядок, надежную последовательность событий. На самом деле, можно пойти и дальше.

Можно предложить разновидность этого жизненного цикла, которая «управляется документами» (document-driven). По завершении каждого этапа должен быть создан документ, который становится основой следующего этапа. Чтобы начать проектирование, требуется спецификация требований. Чтобы начать кодирование, требуется проектная документация. Чтобы начать тестирование, нужна папка с описанием разработанных блоков. Выпуску и сопровождению продукта предшествует отчет о результатах тестирования.

И – о чудо из чудес! – неосязаемое вдруг стало осязаемым. Руководители получили возможность контролировать ход проекта путем изучения этих документов. И едва ли не стали понимать, что в них написано! Вместо клочков бумаги и загадочных листингов в процессе производства программ стали появляться артефакты, доступные для понимания руководства, даже если оно не могло (или не хотело) разбираться в конечном продукте, то есть собственно в коде.

Однако постепенно выяснилось, что происходят нехорошие вещи.

Прежде всего, программисты стали тратить огромное количество времени на составление этих документов. Нельзя сказать, что это время тратилось впустую, потому что документы действительно *способствовали* переходу к очередной стадии жизненного цикла. Но зная, какой именно ас-

пект вашей работы оценивается начальством, вы, естественно, не жалеете времени на то, чтобы придать ему достойный вид. И несколько приукрасить. Программисты поняли, что документация важнее самой программы. Ведь руководство никогда не заглянет в КОД! Важнейшим продуктом процесса разработки программного обеспечения стали документы, знаменующие конец каждого этапа жизненного цикла программы.

Обнаружилась и другая неприятность. Когда появились стандарты на содержание итоговых документов для каждого этапа, программисты увидели, что документы, которые они раньше считали важными, таковыми не являются. По крайней мере, они не столь высоко ценились по меркам руководства или по определениям стандартов. Например, стандарты для написания руководства пользователя были весьма поверхностными. А стандартов для написания документации по сопровождению зачастую вообще не существовало. Например, в стандартах Министерства обороны США об этих документах было сказано очень мало. В итоге, составление руководства пользователя и руководства по сопровождению – и раньше проходившее с трудом, поскольку программисты обычно очень не любят писать текст, – разумеется, стало задачей последней очередности, которой уделялось мало внимания. Ведь ясно же, что завершающие каждый этап документы важнее.

Особенно это обеспокоило тех, кто должен был осуществлять сопровождение. С их точки зрения, срок годности программной документации весьма краток. Поскольку программный продукт неизбежно подвергается модификациям, значительная часть документации быстро устаревает, становясь бесполезной. И самыми бесполезными с точки зрения сопровождения оказываются те самые итоговые документы отдельных этапов. Кому, например, нужна папка с описанием разработки блоков (Unit Development Folder), если в ней практически нет данных о том, как предполагается применять или сопровождать программу? Все встало с ног на голову. Специалист по сопровождению получал доведенную до блеска документацию, которая по большей части пылилась на полке, и при этом был лишен необходимых для работы документов. Стоило ли полагаться на данные, согласно которым 80% стоимости программного обеспечения составляло сопровождение? Если это так, то почему основное внимание не уделялось документации, полезной для сопровождении?

Прошло время, и увлечение негибким жизненным циклом, управляемым документами, постепенно угасло. Положение изменилось под влиянием целого ряда сил. Львиная доля этих сил была так или иначе связана с теми, кто осознал порочность этого способа разработки программ. Кто-то из них, заинтересовавшись «быстрой разработкой прототипов», заметил, что возникающий при этом ранний цикл «проектирование/кодирование/тестирование» просто не стыкуется с требованием жестких сроков завер-

шения этапов и больших итоговых документов, доведенных до блеска. Другие стали изучать альтернативные определения жизненного цикла, такие как спирали и замкнутые круги. А прочие – в основном те, кто зарабатывал программированием себе на жизнь и хотел отлично делать то, что умеет, – молча игнорировали указания руководства и продолжали поступать, как считали правильным.

Приглядимся к последней группе повнимательнее. Это те, кого мы бы отнесли к «неуправляемым». Их методы работы далеки от того, что можно назвать «следованием дисциплине». Тем не менее, это люди, которые правильно делают свое дело. Что же тогда происходило? И насколько типична была подобная ситуация?

Что интересно, ответ на второй вопрос дало исследование. Марвин Зелковиц (Marvin Zelkowitz) из Университета Мэриленда изучал работу группы программистов – что они делали и когда [Zelkowitz 1988]. По его просьбе они записывали, в какой стадии проекта находились номинально (то есть по мнению руководства) и чем занимались на самом деле. Он обнаружил, что фактически никто строго не придерживался жизненного цикла. Разработчики, которые номинально изучали требования, на самом деле могли кодировать и тестировать небольшие прототипы. Разработчики, которые номинально занимались проектированием, могли проводить эксперименты, создавая и тестируя фрагменты кода, чтобы проверить техническую выполнимость предлагаемого конструктивного решения. Разработчики, которые номинально занимались кодированием, могли возвращаться назад и даже пересматривать требования (изредка) или конструктивные решения (весьма часто). А разработчики, номинально занимавшиеся тестированием, на самом деле могли заниматься всем вышеперечисленным.

Фактически Зелковиц обнаружил (хотя прямо не сказал об этом), что строгий жизненный цикл программного продукта – это обман, придуманный разработчиками. Начальству они говорили то, что ему хотелось слышать, а сами делали то, что требовалось.

Ответ на первый вопрос («что происходило?») состоял в том, что *вопреки* усилиям руководства эти непослушные, недисциплинированные и неуправляемые программисты хорошо делали свое дело.

С той поры прошло много времени, и жесткий взгляд на жизненный цикл разработки, разумеется, уже давно не в чести. Может быть, программисты и руководители наконец-то пришли к согласию, и дисциплина вкупе с контролем снова воцарились на рабочих местах?

Боюсь, это не совсем так. Подозреваю, что сегодня руководство проводит в жизнь какое-нибудь другое безумство (как тут не вспомнить нынешнюю идею интегрированного инструментария для «автоматизации управления жизненным циклом»!), а программисты – как обычно – делают свое

дело в обход, как надо, игнорируя дурацкие указания руководства, – они, как всегда, «неуправляемы».

Ссылки

ZELKOWITZ 1988 – «Resource Utilization During Software Development», *Journal of Systems and Software*, September 1988; Marvin R. Zelkowitz.

1.5. Имитация программной разработки

Бывает, кто-то умный решит проблему раньше, чем все остальные успеют заметить ее существование.

Именно такому случаю посвящен данный рассказ.

Когда-то, лет двадцать с лишним назад, всех нас, программистов, учили, что нисходящий метод разработки программ, основанный на технических требованиях (requirements-driven), – самый правильный. На верхнем уровне берутся технические требования, и путем иерархической декомпозиции крупная нерешаемая проблема методически разбивается на кучку мелких решаемых, так что в итоге получается аккуратное древовидное определение проблемы и ее решения.

Но затем произошло нечто странное. Один из самых известных специалистов по программированию, Дэвид Парнас (пожалуй, наибольшую славу ему принесла работа о разбиении программ на модули), написал необычную статью, фактически объяснявшую, как имитировать нисходящую разработку при написании проектной документации – там даже были слова «fake it» (подделать)!

Подделывать нисходящую разработку? Зачем это нам?

Прошло время. Работа Парнаса казалась аномалией, никак не согласующейся со всеми остальными его блестящими работами. Надо же – имитировать нисходящую разработку!

Затем в другой части компьютерного леса обнаружилось еще одно весьма странное явление. Билл Кертис (Bill Curtis), возглавляющий группу исследователей в Консорциуме микроэлектроники и вычислительной техники (Microelectronics and Computing Consortium, MCC), приступил к эмпирическому изучению техники, применяемой квалифицированными разработчиками. В своих ранних работах Кертис противопоставил два метода разработки – «контролируемый» и «оппортунистический». Контролируемая разработка выполнялась так, как полагается, то есть в нисходящем порядке и в какой-то мере иерархически. Оппортунистическая разработка целиком определялась вкусами отдельного разработчика и подчас выглядела почти произвольной. На этой ранней стадии исследования всех

заинтересовали такие неорганизованные, своевольные разработчики. Почему они так себя ведут, если некорректность их методов очевидна?

Однако со временем исследователи из группы Кертиса стали обнаруживать новые факты. Все чаще они сталкивались с тем, что хорошие разработчики в действительности не соблюдали правила контролируемой разработки. Исследователи все чаще наблюдали оппортунистическую разработку. Нередко мысль разработчика перескакивала с одной проблемы на другую, обычно более сложную. Если представить, что разрабатываемый проект действительно имеет древовидную структуру, то эти отклонения уводили в дальние части дерева, относительно слабо связанные с тем местом, над которым номинально трудился разработчик.

Что это было? Явление оказалось настолько распространенным, что Кертис стал заявлять: «В отсутствие принуждения процесс разработки оказывается оппортунистическим». Люди ведут себя не так, как, по общему убеждению, должны это делать.

Может быть, мы наблюдаем здесь фундаментальную ошибку в практике разработки ПО? Или это фундаментальная ошибка в наших представлениях о том, как *нужно* разрабатывать ПО?

Но вернемся к теме «имитации», затронутой Парнасом. Напомню, что по каким-то странным причинам он рассказал нам, как имитировать нисходящую разработку в документации (то есть как притвориться, что мы провели нисходящую разработку, хотя на самом деле этого не было).

Все объясняют открытия, сделанные Кертисом. Оппортунистическая разработка, выполняемая хорошими разработчиками, просто не может быть нисходящей. При этом постороннему невозможно проследить неожиданный ход творческого процесса во время разработки. Поэтому мы «имитируем» нисходящее описание такой разработки, ведь иначе тот, кто с ней знакомится, не сможет ее понять.

Если соединить эти две части головоломки – Парнаса и Кертиса, станет ясно, что проблема связана не с практикой разработки, а с ее теорией. Теория была идеализированной и, как оказалось, неверной. Фундаментальная ошибка связана с моделью процесса разработки, а не с ее практическим применением.

Что же остается в итоге? Теоретическая модель разработки, которая устраивает нас в качестве описательной схемы, даже если приходится ее подделывать, и эмпирическая модель разработки, которую Кертис назвал «оппортунистической» и которая устраивает нас в качестве реального инструмента.

Поскольку слово «оппортунистическая» вызывает странные, произвольные, даже негативные ассоциации, мне больше нравится представлять

этот процесс иным образом. Когда мысли переходят с предмета на предмет, чаще всего это нужно для того, чтобы решить сначала сложные задачи и лишь потом – связанные с ними более легкие. И здесь напрашивается некая древовидная структура, которая не развивается сверху вниз под влиянием общих технических требований, а строится на основании градации сложности задач и их взаимосвязи.

Однако с таким представлением связана одна проблема. Одна и та же задача кому-то покажется трудной, а кому-то – легкой. Обычно трудными оказываются те задачи, которые еще не приходилось решать, а опыт у каждого разработчика индивидуальный. И если ориентироваться на сложность задач, то у каждого получится свое дерево, не похожее на другие. Возможна ли в таком случае какая-либо разумная модель разработки на данных принципах?

Мне больше нравится думать, что процедура разработки может и должна быть основана на правиле «сначала сложное, потом простое». Думаю, оно, по крайней мере, частично, отражает результаты эмпирических исследований Кертиса, а «сначала сложное» представляется как нечто более организованное (даже если это не так), чем «оппортунистическое».

Как бы то ни было, здесь важна не терминология, а суть. Разработка – оппортунистический (гибкий, приспосабливающийся) процесс, и это правильно. Представление разработки должно быть нисходящим, и обычно таким и оказывается. А Дэвид Парнас ответил прежде, чем большинство вникло в суть вопроса!

1.6. Гибкое программирование (Agile): гибкость достигла зрелости

Первое издание этой книги вышло в 1995 году, и в то время среди профессиональных программистов преобладало мнение о необходимости строгого соблюдения дисциплины разработки.

Как много изменилось с тех пор! Если первое издание книги, призывавшее пересмотреть отношение к дисциплине, напоминало глас вопиющего в пустыне, то сейчас голоса сторонников гибкости раздаются столь же громко (и даже чаще), чем голоса проповедников дисциплины. Разумеется, я говорю в первую очередь о защитниках гибких методов разработки ПО (Agile).

Движение за гибкое программирование Agile стало, я бы сказал, естественным результатом тех ориентированных на мастерство «неуправляемых» гибких методов, о которых говорилось выше в этой книге. (Agile – это собирательный термин для таких методологий, как «экстремальное программирование» (eXtreme Programming, XP), Crystal Clear, адаптив-

ное программирование (Adaptive Software Development), скрам (Scrum), метод динамической разработки систем (Dynamic Systems Development Method) и функционально-ориентированная разработка (Feature-Driven Development). Наиболее популярным среди них считается XP.)

Действительно ли Agile – порождение всех различных гибких методов разработки программ, о которых вы только что прочли? Чтобы ответить на этот вопрос, взглянем на ценности и принципы Agile, изложенные в «Манифесте Agile» («Agile Manifesto») и прилагаемых к нему «Принципах манифеста Agile» («Principles Behind the Agile Manifesto»). Эти документы были разработаны в 2001 году на конференции в Сноуберде (Юта). Там присутствовало 17 сторонников перечисленных в предыдущем абзаце методологий, которые тогда назывались «облегченными».

Во-первых, есть группа ценностей Agile, утверждающих сравнительную важность ряда факторов программного проекта (сторонники движения Agile включили их в «Манифест Agile», *http://agilemanifesto.org*):

- отдельные личности и взаимодействие важнее технологических процессов и инструментов;
- работающая программа важнее исчерпывающей документации;
- сотрудничество с заказчиком важнее заключения контракта;
- реакция на изменение условий важнее строгого следования плану.

А вот список принципов Agile (*http://agilemanifesto.org/principles.html*):

- отдавать высший приоритет удовлетворению заказчика, которое достигается путем ускоренной и непрерывной поставки продукта;
- допускать изменения в требованиях, даже на поздних стадиях разработки;
- часто выпускать работающие версии продукта;
- поддерживать ежедневное общение коммерческих служб и разработчиков;
- привлекать к работе мотивированных программистов, обеспечивать их всем необходимым и доверять им;
- общаться лично: это самый эффективный и полезный способ передачи информации;
- принять за главный критерий прогресса работоспособное ПО;
- поощрять разумную интенсивность работы, избегая смертельной гонки;
- непрерывно заботиться о техническом совершенстве и хороших проектных решениях;
- поощрять упрощение – умение выявить как можно больше работы, которую НЕ нужно делать;

- поощрять самоуправление в командах: оно повышает качество программ;
- регулярно исследовать возможности усовершенствований.

Любопытно сравнить эти ценности и принципы с тем, что было сказано выше об умельцах/оппортунистах. Ряд сходных черт заметен сразу (например, приоритет людей и программ над процессами и документацией). Но вскоре выясняется, что Agile идет гораздо дальше тех первых идей. Такие аспекты, как «сотрудничество с заказчиком», «реакция на изменение условий», «удовлетворение заказчика» и «допущение изменений в требованиях», оказываются значительно глубже, чем более простые размышления о гибком подходе, имевшие место в 1995 году.

Большинство из того, что движение Agile добавило к первоначальным идеям, в значительной мере связано с социологией программных проектов. Многие цели и принципы касаются отношений с заказчиком. Прочие характеризуют отношения внутри организации, разрабатывающей ПО. Такое введение социологии в понятие о технической гибкости разработки – интересный и важный вклад сторонников Agile.

Один из самых интересных вопросов по поводу Agile-методов – возможность их применения. В каких проектах можно их применять? Когда Agile-методы только появились, ответ на этот вопрос казался очевиден. Элистер Кокберн (Alistair Cockburn) указал желательные места применения Agile-методов в своей ранней книге «Agile Software Development» (Гибкая методология разработки) [Cockburn 2002]). По его мнению, их следовало применять только в проектах, удовлетворяющих таким условиям:

- в одном помещении работают от двух до восьми человек;
- эксперты по применению находятся неподалеку;
- цикл разработки длится месяц;
- есть полностью автоматизированные регрессивные тесты;
- разработчики обладают большим опытом.

С Кокберном согласился даже такой активный сторонник дисциплинированного подхода, как Барри Боэм (Barry Boehm). В своей книге «Balancing Agility and Discipline» (Между гибкостью и дисциплиной) [Boehm and Turner 2004], где была предпринята попытка компромисса между этими двумя подходами, Боэм писал, что Agile-методы следует применять командам, разрабатывающим небольшие системы, когда общение с заказчиками/пользователями ничем не затруднено, а технические требования изменчивы. С другой стороны, Боэм отметил, что дисциплинированные подходы (он назвал их «плановыми») следует применять в проектах, для которых характерны следующие условия:

- большой объем и сложность при относительно устойчивых технических требованиях;
- высокие требования к надежности/безопасности;
- наличие предсказуемого окружения.

Но эти первые представления о границах применения Agile-методов несколько обветшали. Такие сторонники Agile, как Уильямс и Кокберн [Williams and Cockburn 2003] (да, тот самый упомянутый выше Кокберн!), подняли ставку размера проекта до «50 или меньше программистов, работающих в одном месте». Другие практически сняли ограничение на размер проекта: [Sutherland 2001] утверждает, что «Agile можно масштабировать на крупные проекты».

Конечно, здесь решался вопрос о том, насколько широко можно использовать Agile-методы. Если они подходят только для небольших команд, то это весьма сужает область применения (и, заметим, влияет на подготовку специалистов, ибо если Agile-методы применимы только в маленьких проектах, вряд ли стоит включать их в академические программы по компьютерным наукам).

В истории программирования уже случалось, что новые идеи, предлагаемые для определенного узкого применения, неожиданно возвеличивались сверх всякой меры. Например, язык программирования Ada первоначально разрабатывался для поддержки систем реального времени и содержал определенную порцию системных средств, позволяющих писать компиляторы Ada на самом языке Ada. Но вскоре приверженцы Ada стали рекомендовать его для применения во всех предметных областях, в частности, для бизнес-систем, где вовсе не предполагалось его использовать. (Любопытно отметить, что эти чрезмерные притязания стали в итоге предсмертным хрипом. Язык Ada в конце концов если не умер, то оказался близок к смерти, и произошло это отчасти из-за старания использовать его в областях, для которых он не предназначался.)

Выше я упомянул строгие ограничения на область применения. Бдительный читатель может поинтересоваться, хотя бы мимоходом, каковы эти ограничения в действительности. Прекрасный ответ на этот вопрос был дан в интересном споре между сторонниками Agile и дисциплинированных методов, приведенном в [Beck and Boehm 2003]. Боэм (да, все тот же Барри Боэм) привел данные, которые, на первый взгляд, укрепляли позиции сторонников Agile, – о том, что добрых 60% программных проектов можно считать маленькими, поскольку в них участвует меньше 10 человек. Но, как отметил Боэм, оценивая такие данные, нужно учитывать фактическое время, затраченное на проект, которое зависит как от количества участников, так и от количества написанных строк кода. При таком подходе лишь 17% всех программных проектов в мире оказываются

маленькими. Таким образом, по мнению Боэма, роль Agile-методов во всей системе программирования невелика.

Другая проблема применимости Agile-методов связана с вопросом о мере методологической чистоты, которую следует соблюдать их сторонникам. Сторонники Agile-методов обычно рекомендуют принимать их безоговорочно (в отдельных гибких методологиях, вроде экстремального программирования, есть множество конкретных указаний и запретов).

Интересный «практический» ответ на этот вопрос есть в [Lindvall et al 2004], где изучается применение Agile (обычно экстремального программирования) в ряде крупных организаций. Четыре фирмы сообщили об успешном применении Agile (выразившемся, в основном, в росте качества ПО, а также в легкости освоения, увеличении гибкости, улучшении морального состояния и сокращении издержек). Но ни в одном из этих случаев не удалось воспользоваться стандартными способами применения Agile-методов. Во всех четырех компаниях сообщили, что индивидуальная подгонка методов «абсолютно необходима» по таким причинам, как: а) потребность в известных формальностях и документации в проектах, где участвует несколько команд, использующих Agile-методы, б) естественный конфликт таких концепций Agile, как непрерывный рефакторинг и непрерывная интеграция, с большинством способов управления конфигурацией. В основном в этих четырех компаниях такая подгонка требовалась там, где новые методы (Agile) должны были взаимодействовать со старыми (традиционными) подходами.

Между тем, в ходе обострения споров о применимости возникает любопытная перебранка, подчас напоминающая описанную выше стычку между программистами в баре на Прямом шоссе. Сторонники Agile говорят о «вызывающем трепет хаосе» (некоторые находят Agile-проекты хаотичными) и «отбросах структуры» (структурированные методы, под которыми они понимают соблюдение дисциплины, для них скучны и утомительны). Приверженцы дисциплины склонны называть Agile-методы «структурированной анархией».

Кроме того, слышны голоса другой школы. Сковронский [Skowronski 2004] утверждает, что все отмеченные выше социологические факторы несовместимы с ключевой для творческого решения проблем ситуацией, когда над задачей работает единственный исследователь. Попытавшись представить таких блестящих творцов, как Исаак Ньютон и Фома Аквинский, – известных любителей работать в одиночку, – работающими в паре, как это принято в экстремальном программировании, он делает вывод, что это совершенно невозможно. Разумеется, тут мы сталкиваемся с новой проблемой: какова роль индивидуального таланта в современных программных проектах? Этот вопрос, хотя он тесно связан с темой данной книги, мы все же оставим для последующих дискуссий за ее рамка-

ми! (Лично я убежден, что выдающийся талант может быть очень ценным на определенном этапе жизненного цикла, например в проектировании, но в целом мешает решению других задач в крупных проектах.)

Встает и еще один вопрос, также имеющий отношение к данной книге. Способствуют ли Agile-методы усилению творческого начала? В той же книге Сковронский, конечно, отвечает на данный вопрос отрицательно. Но альтернатива Agile-методам – не блестящие индивидуальные таланты, чего так хотелось бы некоторым из нас, а дисциплинированный подход. И с учетом того, что гибкость Agile допускает гораздо больше творческой свободы, чем «отбросы структуры» дисциплинированного метода, я первым соглашусь с теми, кто считает, что Agile-программирование лучше сочетается с творчеством, чем его традиционные альтернативы.

Чем все это закончится? Большинство программных новшеств вызывает непомерный ажиотаж, который утихает с появлением рядовых инструментов программирования, часто полезных, но далеко не соответствующих первоначальным претензиям. Вероятно, есть немало оснований считать, что то же самое произойдет с Agile-методами, как это случилось со структурным программированием (и, пожалуй, с объектно-ориентированным подходом), а ранее – со средствами CASE и языками четвертого поколения.

Но возможен и другой вариант. Все чаще утверждают, что Agile-методы должны занять свое место наряду с традиционными дисциплинированными подходами как альтернативный способ разработки для проектов определенного рода. Например, это стремится показать Боэм в процитированной выше книге.

Ссылки

BOEHM AND TURNER 2004 – *Balancing Agility and Discipline*, Addison-Wesley, 2004; Barry Boehm and Richard Turner.

COCKBURN 2002 – *Agile Software Development*, Addison-Wesley, 2002; Alistair Cockburn.

EVANS AND BOEHM 2003 – «Agility Through Discipline: A Debate», *IEEE Software* special issue on Agile methods, June 2003; Kent Beck vs. Barry Boehm.

LINDVALL ET AL 2004 – «Agile Software Development in Large Organizations», *IEEE Computer*, Dec., 2004; Mikael Lindvall, Dirk Muthig, Aldo Dagnino, Christina Wallin, Michael Stupperich, David Kiefer, John May, and Tuomo Kahkonen.

SKOWRONSKI 2004 – «Do Agile Methods Marginalize Problem-Solvers?» *IEEE Computer*, Oct., 2004; Victor Skowronski.

SUTHERLAND 2001 – «Agile Can Scale: Inventing and Reinventing SCRUM in Five Companies», *Cutter IT Journal* special issue on the Great Methodologies Debate: Part 1, Nov., 2001; Jeff Sutherland.

WILLIAMS AND COCKBURN 2003 – «Agile Software Development: It's About Feedback and Change», IEEE Software special issue on Agile methods, June, 2003; Laurie Williams and Alistair Cockburn.

1.7. Странный случай с карандашом корректора

Иногда война между дисциплиной и свободой опускается до абсурдной мелочности. Вот пример – карандаш, вставленный за ухо.

Случилось это во время моей учебы в колледже, когда я, как и большинство студентов, чтобы свести концы с концами, устроился летом на работу, далекую от профессии, которой я собирался овладеть. Я был корректором, то есть вычитывал гранки до того, как они превратятся в газету, которую приносят вам домой, и должен был находить в них ошибки (первая оплачиваемая отладка в моей жизни!). Газета называлась «Quincy Illinois Herald-Whig», то есть была типичной консервативной газетой маленького среднеамериканского городка.

В вечернюю смену трудились два корректора. Мы должны были проверять гранки для утреннего выпуска, который доставлялся сельским подписчикам газеты ночной почтой. Газета, по сути, была вечерней, и разносчики доставляли *первый* выпуск жителям Квинси еще до того, как вечерняя смена приступала к работе. Поэтому работа в вечернюю смену не была престижной – ни мы, временные летние работники, ни наш босс не могли рассчитывать на скорое продвижение по службе.

Однажды вечером я, как обычно, сидел за корректорским столом, бдительно выискивая вездесущие ошибки в оттиске, лежавшем передо мной в ярком свете лампы. Наборщики работали неплохо, и ошибки попадались далеко не в каждом абзаце. Корректорский карандаш я по привычке заложил за правое ухо, чтобы выхватить его и пометить ошибку, как только ее обнаружу.

Мой босс, стиль управления которого заключался в грубых командных окриках и бесцельном шатании, обходил дозором свои крохотные вечерние владения. С сердитым видом он подошел ко мне. У него всегда такой вид, но именно *сегодня* его особый гнев, казалось, был направлен прямо на меня. Я почуял недоброе.

Мое опасение оправдалось мгновением позже, когда он, упершись в меня взглядом, прорычал, чтобы я вынул карандаш из-за уха и держал его в руке над гранками, как положено.

Я пришел в смятение. Первым побуждением было сообщить ему, куда *он* может поместить мой карандаш, но я быстро сообразил, что такое предложение только усилит конфронтацию. Потом я подумал, что не стану обращать внимание на его указание... но и так поступить было, конечно же, невозможно. Наконец, я решил, что откажусь выполнить его требование. Однако создавать серьезную проблему из-за того, где должен находиться мой карандаш во время работы, было бы нелепо, и я осознал, что разумнее всего сделать то, что он требует.

Карандаш переместился из-за уха в руку, при этом, внутренне кипя, я с ответной злобой взглянул на него, чтобы он знал, что я думаю по поводу его указания. Даже тогда, будучи юным студентом, я понимал всю нелепость его требования и что еще большей нелепостью было бы вступить с ним в пререкания.

Размышляя над этим инцидентом в последующие годы, я пришел примерно к такому выводу: граница между приемлемыми и неприемлемыми стилями управления очень тонка. Руководитель, безусловно, должен нести ответственность за продукт, производимый его подчиненными, то есть в какой-то мере и за то, каким образом они его производят.

При этом руководитель отвечает за продуктивность своих подчиненных, производящих продукт по определенной технологии. Продуктивность зависит от многих обстоятельств, среди которых важное значение имеет ощущение работником того, что для руководства ценны он сам и его мнение. Ощущение благополучия и ценности своего труда значительно больше способствует росту продуктивности, чем многочисленные технологические усовершенствования, особенно если они спущены сверху, а не являются результатом личного творчества.

Тот руководитель был просто не прав, когда занимался микроадминистрированием на уровне моего карандаша, но подобную ошибку допускает не он один. Заполонившие компьютерный мир призывы к дисциплине – новые методологии, новые попытки автоматизировать производственный процесс, новые структуризации – все чаще приводят к тому, что вовлеченные в этот процесс руководители пытаются решать проблемы разработки, указывая место, где корректор должен держать свой карандаш. А пользы от этого никакой.

Дисциплина, разумеется, – необходимый элемент процедуры разработки больших программ, но в разумных пределах. Дисциплина – *не* тот показатель, рост которого всегда и безусловно желателен. Она как приправа, которой следует пользоваться умеренно. И босс, управлявший моим карандашом, как и руководители, принуждающие своих работников применять сомнительные «убойные решения» в программировании, явно пересолил нанесенную им рану.

1.8. Индекс сложности

В этой книге мы уже познакомились со множеством разнообразных проблем и решений в области разработки программного обеспечения. В этой главе мы уже обсуждали два важных, но очень разных подхода к решению этих проблем. Один из них предполагает дисциплину, а другой – гибкость. Как постепенно выясняется, допутимо и сочетать эти два подхода.

Все это звучит несколько абстрактно и загадочно. Но чего мне хотелось избежать, так это скучно-абстрактного и скучно-загадочного обсуждения предмета, который в действительности не является ни слишком скучным, ни слишком абстрактным, ни слишком загадочным. Вот почему я был несказанно рад, когда П. Дж. Плогер (P.J. Plauger), один из моих любимых авторов, пишущих на компьютерные темы, изобрел удачный термин для обсуждения всех этих тонких различий [Plauger 1992]. Все это он весьма неформально называет «индексом сложности» («the falutin' index»).

Термин «falutin'» (как в «high-falutin'» и «low-falutin'») он применяет к разным связанным понятиям, означающим сложность проблемы и ее решений. На американском сленге словом high-falutin' характеризуют нечто сложное и изощренное. Напротив, low falutin' означает нечто простое и безыскусное.

К чему же Плогер применил свой «индекс сложности»? К целому ряду вещей:

1. Задачи могут иметь высокую или низкую сложность. Иными словами, они могут быть невообразимо сложными или совершенно простыми.
2. Решения тоже могут иметь высокую или низкую сложность.
3. Люди тоже могут характеризоваться высокой или низкой сложностью.
4. Наконец, программные проекты могут иметь высокую или низкую сложность.

Все это могло бы остаться просто игрой в слова, но Плогер высказал более серьезные мысли. Он считает, что крупнейшие катастрофы в разработке программ происходят при несоответствии разных аспектов проекта по степени сложности. Например, когда для решения сложных задач предлагаются простые методы (или наоборот). Либо программисты высокой сложности работают над проектами низкой сложности (или наоборот).

Он привел несколько известных ему случаев такого несоответствия. «Однажды я увидел встроенную систему, которая работала просто-напросто как торговый автомат, – пишет он. – Ее функции ограничивались тем, чтобы подсчитать опущенные монеты, отпустить товар и дать сдачу». Далее Плогер пишет, что разработчик применил здесь подход высокой сложности для решения задачи низкой сложности. «В предложенном конст-

руктивном решении обработка данных осуществлялась с помощью шести отдельных процессов, выполняемых под управлением коммерческой операционной системы реального времени. Не сомневаюсь, что разработчик с удовольствием применил свои специальные знания, но он явно перестарался».

В итоге проект высокой сложности пришлось отвергнуть. Финальная система применяла обычный цикл с опросом и гораздо меньше аппаратуры, чем требовало громоздкое, изощренное решение. Иными словами, сложный программист попытался решить простую проблему в простом проекте сложным способом. А это нехорошо. Совсем нехорошо!

Проблемы могут возникать и в случае обратного несоответствия. Плогер рассказывает о важном проекте, в котором сложные задачи пытались решить с помощью простых средств. Он пишет: «Это болезнь больших программистских компаний, управляемых «вертикально», где руководству удобно считать программистов товаром. Там нанимают неквалифицированных программистов... которых потом если и обучают, то очень мало».

«Руководство пытается управлять процессом, – пишет он далее, – требуя частых отчетов о ходе работ, рецензируя код и организуя такого рода контроль качества, при котором еще более неквалифицированные программисты пишут простые контрольные примеры для проверки простого поставляемого кода. Результат нетрудно предвидеть. Наверняка вы сами или кто-то из ваших знакомых наблюдали подобную катастрофу либо читали о подобной финансовой неудаче». Как только заходит речь о сложном решении простой проблемы, или наоборот, ждите крупных неприятностей. К тому, что рассказал Плогер, многие программисты наверняка добавят собственные истории о несоответствии сложности задач и решений.

Отмечу еще один важный вывод, сделанный Плогером в этой связи. Не существует единого подхода с «правильной» сложностью. Реальный мир ставит задачи как высокой, так и низкой сложности. Литература по программированию предлагает как простые, так и сложные методы (и программисты должны осознавать степень их сложности). Одни программисты предпочитают работать в условиях низкой сложности, другие – наоборот, высокой. Здесь нет правых и виноватых. Есть лишь различия, и эти различия нужно учитывать, привлекая людей к работе над проектом, чтобы обеспечить наилучшее решение задачи.

«К чему нужно стремиться, – говорит Плогер где-то, – так это к согласованности». По его мнению, применяемые методы должны соответствовать естественной сложности проблемы, которую вы пытаетесь решить.

Выберете неоправданно упрощенный подход – и не уложитесь в отведенные сроки. Или даже вообще не справитесь с проектом. И это классический случай бедствий, случающихся в программировании: сначала суще-

ственно заниженная оценка времени, затем неадекватный выбор методов управления и технологий, а после – лихорадочные попытки завершить проект, гонка на выживание. «Любой достаточно крупный проект, – говорит Плогер, – это, главным образом, испытание методов управления. От технологий программирования конечный результат зависит лишь в незначительной мере».

А что будет, если выбрать неоправданно усложненный подход? Вы потеряете деньги и время – и немало. Составление документации и отчетов поглотят вас настолько, что трудно будет определить, решена ли задача удовлетворительным образом.

Итак, применение «индекса сложности» – не гладкий процесс. Какой же итог всему этому подводит Плогер? Он заключает: «С учетом всех аспектов количество различных комбинаций степеней сложности практически бесконечно. Вспомните проекты, в которых вы участвовали. Были ли они успешными? Хорошо ли при этом согласовывались показатели сложности? Мой личный опыт показывает, что ответы на эти два вопроса прочно связаны между собой... Успешные проекты гармоничны, иначе они не были бы успешными».

Ссылки

PLAUGER 1992 – «The Falutin' Index», *Embedded Systems Programming*, May 1992, pages 89-92; P.J. Plauger.

1.9. «Странная парочка» – дисциплина и творчество

Некоторые считают дисциплину и творчество «странной парочкой».

Такое название связано с тем, что эти два слова вызывают у нас в мозгу совершенно разные образы. Можно ли их совместить?

Дисциплина наводит на мысль о марширующих шеренгах, движущихся с заданным шагом в указанном направлении.

Творчество наводит на мысль о ярких индивидуальных исследователях, движущихся каждый в своем ритме по собственному пути. Что у них может быть общего?

Если задуматься, то можно заметить, что жизнь неизменно представляет собой сплав обеих тенденций. Исполняя или слушая музыку, можно наслаждаться как дисциплинированным исполнением тщательно написанного музыкального произведения, так и творческой импровизацией исполнителя-виртуоза. Обратившись к научным достижениям, понимаешь, что налагаемая научным методом дисциплина является основой для

неожиданных, иногда кажущихся случайными, но, несомненно, творческих открытий, составляющих абсолютное большинство в том, что дала нам наука.

Если обратить внимание на то, как пишутся стихи, то становится ясным, что ритм поэтических форм скорее направляет, чем ограничивает наш творческий выбор. Скажем, в поисках рифмы мы явно сочетаем дисциплину поиска рифмы (то есть можем отыскивать рифму к конкретному слову, систематически перебирая буквы алфавита и добавляя к ним заданное окончание) и творческую способность к образному мышлению.

Разработка программного обеспечения ничем не хуже: для нее требуется та же «странная парочка». К примеру, творческое проектирование чередуется с дисциплинированным кодированием. Методологии проектирования стремятся превратить его в упорядоченный процесс, но пока будут возникать новые задачи, требующие решения, полностью достичь этой цели не удастся. Место для творчества находится при написании кода, при решении мелких задач, не проработанных в проекте, который никогда не бывает всеобъемлющим, и в процессе выполнения абсолютно необходимого требования написать код, который в мельчайших деталях диктует глупому компьютеру, как реализовать нужное решение.

Особенно характерна эта «странная парочка» для сопровождения программного обеспечения. Программный продукт, требующий сопровождения, налагает огромные дисциплинарные ограничения на творчество специалиста по сопровождению. В процессе добавления новых возможностей нельзя затрагивать текущую функциональность программного продукта. Творить в таких крайне стесненных условиях, пожалуй, труднее всего, и именно по этой причине большинство программистов старается избегать работ, связанных с сопровождением. В сложном мире разработки программ, наверное, нет ничего сложнее попытки остаться творцом в условиях незыблемых дисциплинарных ограничений.

Возникает соблазн – особенно в мире программирования – считать дисциплину и творчество враждебными друг другу. Сторонники дисциплины заявляют, что чем она строже, тем эффективнее разработка программ. Сторонники творчества заявляют, что смягчение дисциплины – залог повышения эффективности разработки. Разгорается сражение, в котором люди встают на ту или иную сторону, упуская из виду то обстоятельство, что ни те, ни другие не могут быть абсолютно правы.

Для будущего прогресса в нашей области необходимо найти правильное сочетание двух противоречивых начал – дисциплины и творчества, и на пути к этому прогрессу оба лагеря должны сложить оружие и начать сотрудничать.

Глава 2 | Формальные методы и эвристики

Введение
2.1. По поводу одной дискуссии
2.2. Программирование без комплекса вины
2.3. Формальные методы: драма (успех, провал)
2.4. За границами формальных методов
2.5. Письма читателей: размышления о формальных методах

Введение

Методом проб и ошибок выполняется огромное количество работы... Постоянный переход от наблюдений к теории и обратно. Без теории неясно, что искать, а без фактов невозможно проверить теорию... Думаю, любое исследование требует тысяч, даже миллионов таких переходов между тем и другим.

Джошуа Ледерберг,
лауреат Нобелевской премии в области генетики

цитируется по книге Горация Фриленда Джадсона
«The Search for Solutions»,
Holt Rinehart and Winston, 1980
с разрешения Henry Holt and Company Inc.

Надеюсь, читатель простит мне склонность к игре слов.

Эта глава, как видно из названия, посвящена формальным методам и эвристикам. Здесь говорится о том, какую роль в создании программного обеспечения играют формальные методы – подход, пропагандируемый в настоящее время большинством специалистов по компьютерной науке, – и альтернативный, почти противоположный подход.

Что же скрывается за этими словами?

В моем словаре «формальный» определяется как «удовлетворяющий принятым правилам или обычаям; систематический или планомерный», а «эвристический» означает «стимулирующий учащегося к самостоятельному добыванию информации; применяющий метод проб и ошибок».

Словарное определение «формальности» не слишком приближает нас к термину «формальные методы», используемому в компьютерной науке, хотя и дает некоторое представление о возможной цели этих методов. Суть формального подхода связана с применением правил и систематичностью. (Встречающиеся в компьютерной науке определения формальных методов несколько различаются, но в целом их роднят две идеи: «формальная спецификация» как описание требований задачи на математически строгом языке и «формальная верификация» как математические методы «доказательства» того, что программа соответствует своей спецификации.) Специалисты по компьютерной науке рассматривают формальные методы как математически обоснованный систематический подход к решению задач.

Суть эвристических методов в том, что они требуют исследований и не являются систематическими. В отличие от дилеммы, рассматривавшейся в предыдущей главе, где я пытался примирить «дисциплину» с противоположным подходом, здесь у нас, как мне кажется, довольно яркий образец очевидных противоположностей.

Особая важность этой пары терминов в книге, посвященной творчеству в программировании, обусловлена тем, что, как мы видели в предыдущей главе, у каждого из них (и у их основополагающих концепций) есть сторонники, считающие свой способ разработки программ лучшим. Они не только пропагандируют свой выбор, но и за несколько лет инициировали ряд весьма эмоциональных выступлений в прессе, пытаясь добиться доминирования не вполне честными способами!

О каких выступлениях идет речь? В конце 1970-х вспыхнул интерес к «социальным процессам». Три уважаемых автора Ричард Демилло (Richard A. DeMillo), Ричард Липтон (Richard J. Lipton) и Алан Перлис (Alan J. Perlis) опубликовали в нескольких ведущих компьютерных журналах статью, в которой утверждали, что формальные математические доказательства – а следовательно, доказательства в компьютерных науках – не столь абсолютны, как всегда считалось, и что полагаться на доказательства можно лишь тогда, когда они сопровождаются искусно выработанными социальными процессами. Ответ на их выступление последовал почти мгновенно. Знаменитый ученый (и формалист) Эдсгер Дейкстра выступил с резкими возражениями, назвав эту «совершенно безобразную», по его мнению, работу «политическим памфлетом из средневековья» и заявив, что авторы «зашли гораздо дальше обычной негативной реакции практиков».

Вскоре подоспели сторонники вызвавшей шум статьи. В ноябрьском номере «Communications of the ACM» за 1979 год отчаявшиеся прагматики превозносили направленную против доказательств статью в выражениях «лучшая из компьютерных публикаций, которые я читал» или «восхитительно, восхитительно». Один даже заявил: «Пора было кому-нибудь четко и громко сказать: формальный подход к верификации программ не действует и, видимо, никогда не будет действовать...»

(Более полный отчет о происходивших событиях, а также ссылки на соответствующие статьи есть в [Glass 1981].)

Впереди нас ждало еще одно шумное событие. Лет через десять после первой статьи против доказательств и формальных методов появилась вторая, на этот раз Джеймса Фетцера (James H. Fetzer). И снова реакция была скорой и бурной. Десять уважаемых сторонников формальных методов написали письмо редактору «Communications of the ACM», обвинив Фетцера в «искажениях», «некомпетентности в предмете», а также «без-

ответственных и опасных действиях». Фетцер ответил им в том же духе, сказав, что его критики продемонстрировали «полное непонимание проблем» и «слабоумие», а их поведение сравнил с действиями «религиозных изуверов и идеологических фанатиков».

(Более полный отчет об этих событиях, включая соответствующие ссылки, есть в [Glass 1991].)

Почему формальные методы вызывают такую бурю эмоций? Потому что это совершенно иной метод создания программ. Потому что он в значительной мере остается непроверенным и недоказанным. Потому что за прошедшие годы он не нашел особой поддержки и распространения среди действующих программистов. Потому что те, кто уверовал в его мощь, ведут беззастенчивую пропаганду в его пользу и иногда даже навязывают его силой. И потому что его защитники склонны критиковать своих оппонентов.

Выше я уже говорил, что продвижение формальных подходов осуществляется не вполне достойными способами. Во время первого издания этой книги сторонники формальных методов явно одерживали верх, несмотря на яростное сопротивление противников. В учебники компьютерных наук все чаще включался материал по формальным методам как основа для обучения начинающих. К поддержке формальных методов были привлечены органы государственной власти – например, британское правительство требовало применения формальных методов при создании определенных видов программного обеспечения. Большинство преподавателей компьютерных наук занималось обучением формальным методам и их пропагандой, не желая (как будет показано ниже в этой книге) прислушиваться к аргументам о том, что эти методы недостаточно опробованы и проверены. Большинство компьютерных конференций – во всяком случае те из них, что старались угодить академическим кругам, – включали формальные методы в основные темы.

К началу нового тысячелетия практически не оставалось сомнений, что формальные методы достигли высокого официального положения (об этом сказано выше). В начавшемся тысячелетии это положение в значительной мере сохранилось.

Остались ли еще сторонники у эвристических методов? Или они окончательно проиграли сражение?

В другой части нашего компьютерного леса, рядом с более обширным лесом, населенным традиционными научными дисциплинами, есть предметная область, называемая «решением задач» (problem-solving). И можно заметить, что в области решения задач господствуют совсем иные взгляды, чем в компьютерных науках. По мнению ученых, исследующих проблемы решения задач, чем выше сложность задач, тем менее успеш-

ными оказываются формальные методы. Несмотря на более низкий теоретический ранг эвристических методов (то есть методов проб и ошибок), фактом является то, что решать важные задачи – по заявлению данных специалистов – позволяют только эти методы. Один из главных представителей этой точки зрения – Герберт Саймон (Herbert Simon), лауреат Нобелевской премии и премии Тьюринга.

Здесь, разумеется, возникает вопрос: являются ли задачи, решаемые компьютерными программами, сложными и важными? Честным ответом будет, конечно, «в некоторых случаях – да». Но тут обнаруживается важность открытий, опубликованных в [Dekleva 1991]. Автор выяснил, что задачи, которые программисты решали в 1990 году, были в 50 раз крупнее задач 1980 года! На основании этого открытия можно сказать, что даже не слишком сложные задачи, решаемые программным обеспечением, гораздо сложнее по сравнению с такими же задачами десятилетней давности, будь то 1980-е, 1990-е или даже 2000-е.

В данной главе я пытаюсь отыскать хоть какую-то опору в этой пучине эмоций. Согласие с моими выводами, видимо, зависит от вашей изначальной склонности к формальным или эвристическим методам (боюсь, это уже вопрос не теоретических споров, а пропаганды и боевых действий). Тем не менее, постарайтесь во время чтения как можно менее предвзято отнестись к этому жизненно важному вопросу. От его решения может зависеть будущее компьютерной науки.

Ссылки

DEKLEVA 1991 – «Real Maintenance Statistics», Software Maintenance News, February 1991.

GLASS 1981 – «A Look at Other Upstream Paddlers», *Software Soliloquies*, Computing Trends, 1981; Robert L. Glass.

GLASS 1991 – Предисловие к *Software Conflict*, Yourdon Press, 1991; Robert L. Glass (переиздано developer.* Books в 2006 как *Software Conflict 2.0*).

2.1. По поводу одной дискуссии

В компьютерной науке существует тлеющий конфликт на стыке двух важных тем – формальных методов и эвристик. Этот конфликт, который может показаться теоретическим и понятным лишь посвященным, на самом деле затрагивает суть наших представлений о направлении развития программирования.

Некоторые не видят здесь предмета для споров. Многие теоретики компьютерной науки активно пропагандируют формальные подходы и склонны рассматривать эвристические методы как второстепенные. Эта точка

зрения отражена в согласованных тематических выпусках трех ведущих журналов IEEE, относящихся к программированию [Computer 1990, Software 1990, Transactions 1990], где сказано, что результатом применения формальных методов будут «виртуозные программы».

Далее, формальные методы часто составляют суть рекомендаций консультативных комитетов по компьютерным наукам в отношении направлений развития программирования. В отчете технической комиссии по компьютерным наукам Национального научно-исследовательского совета (Computer Science Technology Board of the National Research Council) [CSTB 1990] под заголовком «Укрепление фундаментальной базы» («Strengthen the Foundations») сказано:

> «Из-за отсутствия прочного научного и технического фундамента сложные программные системы часто изготавливаются путем грубого форсирования, когда руководство неуклонно наращивает количество участников проекта, а сроки осуществления проектов неуклонно растут. Разработчики уже приступают к системам, реализация которых требует многих тысяч человеко-лет, и нынешние практические или эвристические подходы оказываются все менее адекватными решаемым задачам. В этих условиях руководители программных разработок начинают прибегать к более систематическим средствам: возникает потребность в математических, научных и технологических методах».

Трудно не согласиться с таким заявлением. В общем-то, все знают, что тонкая сила лучше грубой, прицельная стрельба лучше беспорядочной, а систематический подход лучше эвристического.

Но есть и противоположная точка зрения. И она все чаще высказывается в литературе. Обычно она выражается подспудно, появляясь в статье, написанной на какую-то другую тему. Вот ряд примеров.

В статье о проектировании ПО [Rombach 1990], Г. Дитер Ромбах (H. Dieter Rombach) пишет: «Творческая суть процесса проектирования означает, что многие его аспекты не могут быть формализованы... вообще. Хотя формализация... дает решение в случае более механических процессов... для процессов проектирования она непригодна».

В книге по истории медицинской информатики [Blum 1990] Брюс Блюм (Bruce Blum) высказывается еще более резко: «В тех областях, которые нам хорошо понятны, формулы и статистические выводы лучше всего подходят для представления знаний. Для других задач наиболее мощным подходом оказываются эвристики и когнитивные модели».

Даже в упомянутых выше специальных выпусках журналов IEEE, посвященных формальным методам, высказываются некоторые мелкие сомнения:

> «Большинство формальных методов пока неприменимы для описания крупных программных или аппаратных систем; большинство из них все еще не позволяют определять многие важные ограничения поведения, не относящиеся к функциональности, например, отказоустойчивость и функционирование в реальном времени» (Жанет Уинг (Jeanette Wing), [Computer 1990]).

> «...предшествующие задачи [системный анализ и проектирование] лучше решать менее формальными методами» [Software 1990, Gerhart].

У этих нынешних формулировок, касающихся ценности эвристик и ограничений формальных методов, есть прочные теоретические и исторические корни. В книге «The Sciences of the Artificial» (Науки об искусственном) [Simon 1981]) Герберт Саймон, лауреат Нобелевской премии и премии Тьюринга, рассматривает эту проблему в главе «The Architecture of Complexity» (Архитектура сложности):

> «Чем сложнее и новее задача, тем больше проб и ошибок требует поиск ее решения. Отбор осуществляется с помощью различных эмпирических правил, или эвристик, которые предлагают первоочередные пути исследований и наиболее многообещающие направления».

Значит, сколь ни удивительно, но аргумент в пользу эвристических подходов *есть*. Фактически с ростом сложности задач решать их, по мнению некоторых специалистов, можно *только* с помощью эвристических методов.

Примеры

Поскольку данное обсуждение оказалось несколько умственным и абстрактным, неплохо было бы подкрепить его примером. Я бы хотел привести простой случай из собственной жизни, свидетельствующий о том, что происходит с формальными математическими методами, когда сложность задачи возрастает. Задача, на самом деле, не из компьютерных – это вопрос о том, можно ли погрузить крупную вещь в багажник автомобиля с кузовом «универсал».

В первом приближении багажник имеет форму трапециевидной призмы, и нужно лишь измерить ее стороны и размер груза и, сравнив то и другое, узнать ответ.

Но задача усложняется, когда обнаруживается, что форма багажника – не просто трапециевидная призма. Во-первых, внизу у задней двери эта призма немного сужается, чем ограничивается входной проем. Во-вторых, грани призмы на самом деле несколько искривлены. В-третьих, внутри у призмы имеются разные выступы – арки колес, ручки дверей,

крючки для одежды и так далее. В результате, если габариты груза близки к размерам призмы, математика крайне усложняется. Поверхности должны описываться несколькими функциями. то и дело возникают исключительные ситуации. Для решения задачи нужно найти пересечение сложных поверхностей в трехмерном пространстве. Даже я, бывший математик с опытом проектирования геометрии воздушных судов, не взялся бы за столь сложную задачу.

Увидев, что размеры багажника и груза делают возможность размещения весьма спорной, я отказался от сложного математического подхода и прибег к эвристике. Я подтащил груз к машине, вставил узким концом в проем и затем, по-всякому поворачивая, подтягивая и подталкивая его вперед, обнаружил, что (только) при открытой задней двери остается дополнительное свободное пространство, позволяющее повернуть груз нужным образом.

Фактически это пример эвристического подхода к сложной проблеме. Возможно, с помощью математики и формальных методов задачу в конечном счете удалось бы решить (хотя сомневаюсь, что в поисках математического решения задачи я бы догадался увеличить свободное пространство, открыв дверь!); однако такой подход мог:

- требовать дополнительных сложных навыков;
- требовать алгоритмов, которых нет в наличии;
- требовать компьютерной программы, реализующей эти алгоритмы;
- значительно увеличить расход времени;
- увеличить вероятность ошибок.

Важно отметить, что на практике скорость получения результата может быть не менее важна, чем его качество, а главной характеристикой решения может стать его надежность. Исходя из этого я склонен считать, что в данном примере сложной задачи эвристические методы помогли найти решение, а сделать это формальными методами просто не удалось бы.

Некоторые проблемы из приведенного списка можно проиллюстрировать другим примером, на этот раз из компьютерной литературы. Исследовательская лаборатория ВМС США несколько лет назад финансировала проект, известный под разными названиями: проект А-7 и «проект по сокращению стоимости разработки ПО» (Software Cost Reduction Project). (Последнее название, как вы скоро увидите, заключает в себе иронию.) В результате появился ряд статей о формальных подходах к установлению технических требований для сложных систем, на которые часто ссылаются, и в этом смысле проект был весьма успешным.

Но в другом смысле проект провалился. Первоначально планировалось, что этот проект позволит сравнить формальный метод разработки про-

грамм и некоторый специальный метод, для чего будет повторно разработано ПО для самолета А-7 с применением формальных подходов, а потом будут сопоставлены возможности сопровождения формально разработанного кода и первоначального (специального) кода (путем ввода в тот и другой аналогичных изменений). Осуществление этой части проекта пришлось прекратить, поскольку формальный подход потребовал слишком много времени. В рамках установленного бюджета времени и финансов оказалось невозможным провести эксперимент, который оценил бы преимущества формального подхода.

Один из главных участников проекта сделал вывод, что «программы писать тяжело», то есть разрабатывать сложное программное обеспечение гораздо труднее, чем могли себе представить те, кто прежде разрабатывал только простые программы. (Далее эта история будет изложена более подробно.)

«Тление»

В начале этого раздела данный конфликт был назван «тлеющим». Это слово выбрано неспроста.

Несколько десятков лет назад ситуация была неспокойной, и вопрос формальной верификации программ («доказательство корректности») вызвал полемику в печати [DeMillo 1979, Fetzer 1988]. Противники публиковали статьи, критикующие основы всей концепции, сторонники яростно клеймили своих оппонентов, переходя на личности, все больше людей ввязывалось в перепалку, буквально превращая ее во всеобщую драку. Аргументы сторон имели все признаки религиозных войн.

Примерно тогда же Эдсгера Дейкстру [Dijkstra 1989] подтолкнули принять участие в публичном обсуждении ряда вопросов, касающихся разумности преподавания формальных методов студентам младших курсов компьютерных специальностей. Эта (плановая) дискуссия носила более сдержанный характер, чем ее предшественницы, но результат был таким же: сильная эмоциональная и иногда личная реакция спорящих и последовавших затем читательских откликов.

Эти случаи показывают, что движение за формальные методы в настоящее время погрязло в религиозном рвении и реакционном сопротивлении. Ни одна из сторон, похоже, не усматривает рационального зерна во взглядах противника. Очевидно, что проблема решения сложных задач является центральной с точки зрения развития программирования. Почти столь же ясно, что любые распри, мешающие нам решить эту проблему, должны быть преодолены.

Заключение

Цель этого раздела – показать, что существует конфликт между сторонниками формальных и эвристических методов, что дискуссия, порожденная этим конфликтом, вполне закономерна, и «правота» присуща обеим сторонам, и что вести эту дискуссию следует с меньшим накалом страстей. Поскольку дискуссии о значении формализма обычно начинались энергичными заявлениями его сторонников, за которыми следовали опровержения противников, здесь сделана попытка резюмировать положительные стороны противоположной позиции посредством ссылок, примеров и случаев из жизни.

Следует заметить, что при тщательном изучении материала открывается возможность достижения компромисса. Формальные методы, вероятно, уместны при решении механистических и хорошо понятых проблем или их составных частей; эвристики необходимы для решения более сложных и творческих задач. Возможно, формальные методы помогли бы в организации процесса «селективных проб и ошибок», описываемого Саймоном (например, обеспечив нас правилами и методами для поиска вариантов и осуществления отбора). В данном случае, как и в большинстве серьезных споров, может оказаться, что «правильная» позиция соединит в себе лучшие стороны обеих крайностей.

Ссылки

BLUM 1990 – «Medical Informatics in the United States, 1950–1975», *A History of Medical Informatics,* ACM Press, 1990; Bruce I. Blum.

COMPUTER 1990 – Special issue of *IEEE Computer,* on «Formal Methods – Prelude to Virtuoso Software», September 1990.

CSTB 1990 – «Scaling Up: A Research Agenda for Software Engineering», a report of the Computer Science Technology Board of the National Research Council, *Communications of the ACM,* March 1990.

DEMILLO 1979 – «Social Processes and Proofs of Theorems and Programs», *Communications of the ACM,* May 1979; Richard A. DeMillo, Richard J. Lipton and Alan J. Perlis.

DIJKSTRA 1989 – «On the Cruelty of Really Teaching Computing Science», *Communications of the ACM,* December 1989; Edsger W. Dijkstra.

FETZER 1988 – «Program Verification: The Very Idea», *Communications of the ACM,* September 1988; James H. Fetzer.

ROMBACH 1990 – «Design Measurement: Some Lessons Learned», *IEEE Software,* March 1990; H. Dieter Rombach.

SIMON 1981 – *The Sciences of the Artificial,* MIT Press, 1981; Herbert A. Simon.

SOFTWARE 1990 – Special issue of *IEEE Software,* on «Formal Methods – Developing Virtuoso Software», September 1990.

TRANSACTIONS 1990 – Special issue of *IEEE Transactions on Software Engineering* on «Formal Methods», September 1990.

2.2. Программирование без комплекса вины

Допускает ли ваш подход к разработке ПО широкое применение метода проб и ошибок? Не кажется ли вам, что это не лучший способ решения задач? Не подумываете ли вы над тем, чтобы освоить все эти формальные методы компьютерных наук, о которых столько говорят?

Вполне возможно, что на все эти вопросы вы ответили утвердительно. Да, пробы и ошибки встречаются часто. Да, похоже, что это плохой подход. Да, вероятно, нужно попробовать формальные методы.

Однако это лишь один из возможных вариантов ответов. Другой возможный ответ, даже если вам это покажется странным, будет в целом противоположным. И этот ответ означает, что в методе проб и ошибок нет ничего плохого. Есть успешные люди, которые пользуются им постоянно. На самом деле, чем больше вы совершаете проб и ошибок, тем больше у вас шансов на успех.

Я расцениваю такое отношение к созданию программ, как «отсутствие комплекса вины». Чувство вины, испытываемое в отношении метода проб и ошибок, по большей части неоправданно и неуместно.

Остановимся же на какое-то время на применении методов проб и ошибок при создании ПО, уделив особое внимание значению неудач для достижения успеха.

Я вижу тонкую иронию в том, что те, кто добился успеха, обычно ошибаются чаще прочих.

Это странное явление можно наблюдать в разных местах и в разных видах. Например, в программировании оно обнаруживается при сравнении работы успешных работников с работой не слишком успешных: в [Vitalari 1983] показано, что хорошие системные аналитики выдвигают и отвергают больше гипотез, чем плохие. В [Curtis 1987] показано, что хорошие проектировщики применяют метод проб и ошибок для создания мысленных моделей конструктивных решений, проверяя их корректность путем мысленной эмуляции, а плохие проектировщики часто останавливаются на некотором решении, делая меньше проб и ошибок, чем хорошие проектировщики. В [Soloway 1988] показано, что хорошие специалисты по

сопровождению многократно повторяют «эпизоды расследования»: просматривают фрагмент кода, имея в виду некоторый вопрос, предполагают некоторый ответ, а затем ищут в документации и/или коде подтверждение своего ответа, и если найденная информация опровергает предположение, то начинают итеративный процесс снова.

У хороших системных аналитиков, проектировщиков и специалистов по сопровождению есть ряд черт, которыми они отличаются от менее успешных коллег, и здесь мы видим одну из таких черт – способность ошибаться и восстанавливаться после неудачи. Такого рода пробы и ошибки мы называем эвристическим процессом, и наше открытие можно сформулировать иначе: хорошими работниками в программировании являются те, кто овладел эвристическими процессами.

Конечно, это относится не только к тем, кто занимается программами. Есть даже целая книга, посвященная значению провалов: [Petroski 1985] многократно повторяет, что нельзя добиться успеха, не потерпев сначала поражения. В этой книге автор упорно вдалбливает свою мысль названиями глав:

«Падение – этап роста»

«Успех – это предвидение провала»

«Когда трещина ведет к прорыву».

И многими цитатами:

> Инженер не хуже поэта видит недостатки своих произведений и больше учится на своих и чужих ошибках, чем на всех шедеврах, созданных им самим и его коллегами.
>
> Есть большая практическая польза в том, чтобы совершить несколько ошибок в начале своей жизни (цитата из «On Medical Education» T. H. Huxley).

И даже замечательными историями:

> Некоторые учителя сначала были против карандаша с резинкой на конце, поскольку «чем легче исправлять ошибки, тем чаще их будут делать» (цитата из «The Pencil» Petroski).

Помимо того что провалы имеют большое значение, они еще и случаются часто. Вот что говорит о научном методе лауреат Нобелевской премии в области генетики Джошуа Ледерберг: «Методом проб и ошибок выполняется огромное количество работы... Постоянный переход от наблюдений к теории и обратно. Без теории неясно, что искать, а без фактов невозможно проверить теорию... Думаю, любое исследование требует тысяч, даже миллионов таких переходов между тем и другим» [Judson 1980].

Можно даже указать образцы исключительного успеха из прошлого столетия – например, президент Гарри Трумэн или выдающийся производитель сластей Милтон Херши, – и отметить, что годами и даже десятилетиями их преследовали неудачи, включая такие суровые испытания, как банкротство, в конечном итоге обратившиеся в успех, который и сделал этих людей знаменитыми.

Думаю, у подобного объяснения пути к успеху через провал есть пара слабых мест. Во-первых, эта тема граничит со слащавостью. Слишком просто влиться в хор поющих «если не сразу добился успеха, пробуй и пробуй, пока повезет» на мотив «Синей птицы счастья»; и все это, конечно, попадает в разряд «избитых истин».

Другой опасный уклон – начать оправдывать этим скопление ошибок в вашей работе. Одно дело – неудача, постигшая вас в погоне за успехом, и другое – когда она результат лени. В первом случае ничего страшного, во втором – это отговорка.

Значит, когда вы в следующий раз почувствуете вину за то, что метод проб и ошибок при поиске решения стал причиной стольких ошибок, вспомните, что путь к успеху почти всегда ведет через ошибки – и порой многочисленные.

Итак, да здравствует метод создания программного обеспечения, не вызывающий чувства вины!

Ссылки

CURTIS 1987 – «Empirical Studies of the Design Process: Papers for the Second Workshop on Empirical Studies of Programmers», MCC Technical Report Number STP-260–87, September 1987; Bill Curtis, Raymonde Guindon, Herb Krasner, Diane Walz, Joyce Elam, and Neil Iscoe.

JUDSON 1980 – *The Search for Solutions,* Holt, Rinehart, and Winston, 1980; Horace Freeland Judson.

PETROSKI 1985 – *To Engineer Is Human – The Role of Failure in Successful Design,* St. Martin's Press, 1985; Henry Petroski.

SOLOWAY 1988 – «Designing Documentation to Compensate for Delocalized Plans», *Communications of the ACM,* November 1988; Elliot Soloway.

VITALARI 1983 – «Problem Solving for Effective Systems Analysis: An Experimental Exploration», *Communications of the ACM,* November 1983; Vitalari and Dickson.

2.3. Формальные методы: драма (успех, провал)

У одного из самых известных программных проектов всех времен в действительности самая необычная история, о каких я только слышал. Вот история этого проекта.

Странности начинаются уже с его названия. Их два, и выбор зависит от того, как смотреть на историю. (Что не так уж необычно... Гражданскую войну на юге США называли «Войной между штатами», а в то время, когда Великобритания отвоевывала Фолклендские острова, Аргентина сражалась за Мальвинские.) Те, кто считает проект успешным, называют его «Проектом по сокращению стоимости разработки ПО» («Software Cost Reduction project»). Те, кому результаты проекта представляются в ином свете, зовут его просто «Проект А-7».

Независимо от названия проект начинался большими упованиями, обратившимися в итоге оглушительным успехом или катастрофическим провалом – в зависимости от того, кто на что уповал. Поясню подробнее.

Что бы там ни говорили, это был необычный проект. Начать надо с того, что самолет военно-морской авиации А-7 оборудован программным обеспечением для управления его радиоэлектронным оборудованием в реальном времени. Это программное обеспечение трудно сопровождать, оно плохо документировано и втиснуто в компьютер, недостаточно мощный для решения такой задачи. Короче, довольно типичный случай встроенной компьютерной системы.

Известность этого проекта никак не связана с тем, какие задачи решало это ПО. Она связана с тем, что военно-морские силы решили использовать этот проект как полигон для опробования более эффективных методов создания подобных программ.

К делу был привлечен известный компьютерный специалист Дэвид Парнас (David Parnas) с группой помощников-исследователей. Средства выделила Исследовательская лаборатория ВМС. Решено было переписать все программное обеспечение для А-7 с нуля, применив новые прогрессивные методы программирования. Возникла идея превратить проект в гигантский эксперимент и сравнить эффективность новых и старых методов.

Вот как планировалось организовать дело. Парнас с коллегами получают технические требования для ПО А-7 и начинают работу с нуля, не пользуясь имеющимися наработками. Пока они разрабатывают новую версию, старая продолжает эксплуатироваться и сопровождаться. Регистрируются все изменения, внесенные в старую версию программы, а также потраченное при этом время. Когда новая версия ПО сможет выполнять все функции старой версии, какими они были на момент начала эксперимента, в новую версию будут внесены те же изменения, что и в старую. При

этом также будут учтены ресурсы, потребовавшиеся для модернизации. После этого сравнивается стоимость внесения изменений в ту и другую версии.

Короче, планировался самый крупный и полный экспериментальный исследовательский проект в области программных разработок из когда-либо проводившихся. Мало кто сомневался в характере результатов – большинство было уверено, что новую версию будет гораздо проще и дешевле модифицировать, вопрос был лишь в том, *насколько* дешевле.

Однако почти сразу возникли неприятности. Простая фраза из предыдущего абзаца «получают технические требования» обернулась чрезвычайно сложной проблемой. Разумеется, требования были воплощены в имеющемся коде, но это не значит, что в них можно было легко разобраться. Изначальные источники требований – ответственные разработчики или проектная документация – отсутствовали или были недоступны, либо их данные устарели в силу последующих изменений. Фактически, программное обеспечение А-7 вело собственную жизнь, оставив далеко позади исходную документацию и людские знания.

(С годами мы, разработчики ПО, стали понимать, что такая проблема встречается не столь уж редко. Многие создатели информационных систем, пытавшиеся заново переписать старую программу, к своему изумлению, если не ужасу, обнаруживали, что фактически невозможно вычленить задачу из ее программной реализации. Каким бы обоснованным ни казался этот подход на первый взгляд, очень часто оказывается невозможным реконструировать работающую программу [Glass 1991]!)

Однако Парнас и его люди продолжили борьбу. Они всеми способами пытались получить старые технические требования. Они также стали придумывать средства для регистрации требований и последующей работы с ними. В этом, собственно, и усматривают успех «Проекта по сокращению стоимости разработки ПО». На формальные методы регистрации требований и ссылок на них указывают, вероятно, столь же часто, как на все остальные работы последних лет в этой области вместе взятые [Heninger 1980].

Но шло время, и достижение первоначальной цели проекта выглядело все сомнительнее. Не забывайте, что согласно начальным предположениям новый программный проект в известном смысле должен был «догнать» старый, после чего планировалось провести часть эксперимента, посвященную сопровождению. Вместо этого новый проект отставал от старого. Какими бы замечательными ни казались новые формальные методы регистрации требований, цели первоначального проекта оставались недосягаемыми. Если новое программное обеспечение не в состоянии выйти на уровень старого, провести эксперимент просто невозможно.

Прошло еще некоторое время. Успех фазы «сокращения стоимости разработки» начал бледнеть. С точки зрения «Проекта А-7», вся затея стала напоминать бессмысленную трату государственных средств. Даже Дэвид Парнас, обсуждая эту работу, высказал свою неуверенность в конечном результате – станет ли она замечательным исследовательским проектом или кандидатом на премию «Золотого руна» (гипотетическая награда, присуждаемая Конгрессом США тому, кто потратил крупную сумму бюджетных денег самым идиотским образом).

В конечном счете, финансирование проекта было прекращено. Тот эксперимент, который был первоначальной целью проекта, так никогда и не начался.

Поразительно в этой истории то, что бытуют два совершенно противоположных мнения по поводу его результатов. С одной стороны, с гордостью указывают на проделанную работу по анализу и представлению технических требований. С практической точки зрения, это действительно впечатляющая попытка создать и/или восстановить порядок в проекте, который частично вышел из-под контроля. С научной точки зрения, была заложена основа для последующей работы по анализу технических требований в целом.

Однако с другой стороны, нетрудно было встревожиться тем, что не удалось получить результаты, связанные с первоначальными целями проекта. Никаких экспериментальных результатов не было получено вообще. Никаких количественных данных относительно сопровождаемости программного обеспечения получено также не было. Старое программное обеспечение А-7 по-прежнему использовалось, а новое – нет.

Не было единого мнения даже по поводу полезности того, что действительно удалось сделать. У Дэвида Парнаса встречаются высказывания вроде «Сотрудники контроля качества морских разработок [воспользовались новой формулировкой требований], чтобы прогнать свои тесты и убедиться, что они ничего не пропустили... Военно-воздушные силы взяли наш документ по А-7 и переписали его так, чтобы он охватывал их собственное программное обеспечение... Теперь они пользуются им при заключении контрактов, чтобы быстро составлять надежные технические требования при заключении контрактов на модернизацию» [Zvegintzov 1988].

Однако один из основных участников первоначального проекта А-7 однажды отвел меня в сторону во время обеда и сказал: «Мы не смогли применить этот метод Исследовательской лаборатории ВМС в производстве; мы были вынуждены от него отказаться».

Вот так обстоит дело. Две совершенно разные оценки результата. То ли формальные подходы в «Проекте сокращения стоимости разработки ПО»

совершили замечательное дело по приведению хаоса к порядку, то ли – по мнению других – для осуществления этих подходов в «Проекте А-7» потребовалось столько времени, что вся связанная с ними работа была проделана впустую, а методы оказались попросту непригодными для дальнейшего применения.

Успех или провал? Ответ может зависеть от того, кем вы себя считаете – исследователем или практиком. Но как бы вы ни оценивали результат, история этого проекта с неоднозначным названием и неудачным концом прекрасно характеризует дилемму применения формальных подходов или их практических альтернатив.

Формальная строгость, как мы видим, это то, без чего в одних случаях просто не обойтись, а в других случаях – то, чего нельзя себе позволить. Весь фокус в том, чтобы определить, в какой из этих двух ситуаций вы находитесь в данный момент!

Ссылки

GLASS 1991 – «The Re-Engineering Decision: Lessons from the Best of Practice», *The Software Practitioner,* May 1991; Robert L. Glass.

HENINGER 1980 – «Specifying Software Requirements for Complex Systems: New Techniques and Their Applications», *IEEE Transactions on Software Engineering,* January 1980; K.L. Heninger.

ZVEGINTZOV 1988 – «Parnas Interviewed at ESP-2», *Software Maintenance News,* March 1988; Nicholas Zvegintzov.

2.4. За границами формальных методов

Несколько лет назад вышли две книги, посвященные вычислениям. Одна называлась «Наука программирования», другая – «Искусство программирования». Они вышли с промежутком в несколько месяцев. Возможно, оказалось случайным, что они представляли собой вызов друг другу. Кто был прав? Что есть программирование – наука или искусство?

Годы идут, и та же пьеса разыгрывается снова. Та же тема, другие исполнители. Два ведущих докладчика выступают на ежегодной Международной конференции по программированию. Один подчеркивает важность формальных подходов при создании программного обеспечения. Второй подчеркивает важность подходов, основанных на потребностях приложения. Снова возникает вопрос: кто из них прав? Нужно ли разрабатывать приложение согласно формальным математическим правилам или согласно правилам, диктуемым самим приложением?

Проходит еще несколько месяцев. Из недр статьи, написанной по совсем другому поводу, вдруг всплывает все та же проблема. В [Grudin 1989] читаем: «Часто не замечают конфликта между поиском формальных свойств [пользовательских] интерфейсов с одной стороны и проектированием и анализом задач „в интересах пользователя“ – с другой... Чем сильнее стремление [к формальным, единообразным интерфейсам], тем менее содержательны результаты».

Со временем проблема принимает другие формы. Чаще всего в формальных методах усматривают решение проблем программирования, вызванных «недисциплинированными, эвристическими подходами». «Эвристики» начинают представляться воплощением зла или пугалом, которое выставляют в свою защиту формалисты.

Затем неожиданно доносится новый голос. Ученый и активист в области компьютерных наук с большим стажем Питер Деннинг (Peter Denning) в редакционной статье [Denning 1991] провозглашает сущую ересь. «Ничего дурного в формализме нет, – довольно невинно начинает он. – Он продемонстрировал немалую технологическую силу». Но затем следует сюрприз: «Должен сказать, что он ограничивает наши возможности. Мы должны выйти за рамки формализма и изучать, как происходит обмен данными в наших организациях и в наших программных конструкциях... Настало время уделить больше внимания тем туманным, неточным, не поддающимся формализации областям, с которыми мы сталкиваемся изо дня в день».

Во многих научных заведениях корни компьютерных наук тесно переплетаются с корнями дерева дисциплин, называемого «чистой математикой». А центром мира чистой математики являются аналитические и формальные методы, и к так называемой «прикладной математике» там относятся как к нелюбимой падчерице. Такая же точка зрения в силу исторических причин передалась компьютерным наукам.

Но уже подрастает двоюродный брат этой падчерицы, которого зовут «программирование», и между ним и компьютерными науками такие же отношения, какие были между прикладной математикой и чистой. И для этого братца хорошо то, что работает, вне зависимости от строгости. Поскольку сложность задач, решаемых в программировании, быстро растет – по некоторым оценкам увеличивается в 50 раз каждые 10 лет! – мы видим, к примеру, задачи, которые решаются методами моделирования (а стало быть, эвристически), но слишком сложны, чтобы их можно было решить аналитическими (и, следовательно, формальными) методами. Возможно, с ростом мощности компьютеров наши задачи окажутся не под силу формальным методам.

И, наоборот, с ростом сложности решаемых задач перед нами встают такие задачи, единственный способ решения которых – формализация, и только мощным компьютерам под силу реализовать эти методы!

Суть спора ясна. Одни говорят, что сложные задачи можно решать только формальными методами. Другие утверждают, что формальные методы оказываются непригодными для решения масштабных сложных задач.

Но есть еще третьи – и я подозреваю, что когда рассеется дым, станет ясна их правота, – которые говорят, что ответ растущей сложности находится «за границами формальных методов». Изучением того, что подразумевается под этими границами и что находится дальше, мы сейчас и займемся.

Ссылки

DENNING 1991 – «Editorial», *Communications of the ACM*, March 1991; Peter J. Denning.

GRUDIN 1989 – «The Case Against User Interface Consistency», *Communications of the ACM*, October 1989; Jonathan Grudin.

2.5. Письма читателей: размышления о формальных методах

Как только становится известно, что ваши взгляды на некоторую популярную тему противоречат общепринятым, вы становитесь объектом притяжения единомышленников. Мое мнение относительно формальных методов, которое я высказывал на протяжении ряда лет разными доступными мне способами, вызвало ряд интересных откликов.

Несколько лет назад я написал Джеффу Оффуту (Jeff Offutt), бывшему тогда профессором университета Клемсона (а позднее – университета Джорджа Мейсона), выразив свою озабоченность тем, что сильное связывание (характеризуемое, среди прочего, отказом от глобального или общего доступа к данным), хотя и пропагандируется большинством специалистов по компьютерным наукам, но на практике мало кем из них применяется в разработке программного обеспечения. Я выбрал Оффута в качестве адресата, потому что в тот год он выступил с рядом материалов по связыванию на одной из конференций. Оказалось, однако, что я неправильно понял тему его выступления: связывание (coupling), обсуждавшееся среди тестировщиков, к которым обращался Оффут, подразумевало совсем не то, о чем заботился я.

Несмотря на мою ошибку Оффут подхватил тему, которая меня интересовала, и высказал ряд неожиданных мыслей:

> «Поднятый Вами вопрос о защите сильного связывания теми, кто фактически применяет слабое связывание, наводит на многие мысли. Можно даже взглянуть шире и спросить себя, почему мы все время учим студентов и практиков, что нужно писать программы тщательно взвешенным структурированным образом, а запершись в своих лабораториях, пишем программы на C, в которых перемножаем указатели. Я и сам грешу тем, что не всегда следую собственным рекомендациям: будучи аспирантом, я написал многие части системы мутационного тестирования Mothra, не составляя никаких требований и спецификаций, и мои проекты обычно были небрежно записаны на клочках бумаги. Очевидно, в наших формальных методах разработки программ есть какая-то фундаментальная ошибка[1], и мне кажется, что это противоречие со связыванием – одно из ее проявлений».

Другой мой корреспондент, на этот раз практик, Дуглас Кинг (Douglas King) из корпорации Hughes тоже пожелал поделиться со мной некоторыми мыслями. Его больше интересовало сравнительное значение качества разработчиков и технологического процесса при создании программ. Однако по поводу формальных методов он сказал следующее:

> «В некоторых работах Герберта Саймона упоминается „закон планирования Грешема“:
>
> „[Закон Грешема] гласит, что обыденная деятельность стремится вытеснить новую. Если обязанности руководителя предполагают принятие решений как рутинными способами, так и нестандартными, то первые получат преимущество в ущерб вторым“ [Simon 1965].
>
> Под „рутинной“ (programmed) работой Саймон понимает решение известных задач с помощью систематической структурированной процедуры, такой как формальный метод. Под „нестандартной“ (unprogrammed) работой он понимает решение слабо изученных задач с помощью плохо структурированного метода, такого как эвристика. По моему мнению, закон планирования Грешема указывает на то, что *конфликт между формальными и неформальными методами кроется в человеческой природе, и потому никогда не может быть разрешен окончательно*. При всем нашем безумии всегда приятнее дополнительно иметь еще какой-то метод, но такая возможность есть не всегда. В конце концов, методы – это лишь посмертное рациональное обобщение безумия. Любопытно, что хотя систематические процедуры дают нам возможность решать все новые и новые задачи, не видно кон-

[1] Выделения в приведенных цитатах – мои собственные, а не авторов цитат.

> ца появлению таких задач, которые им неподвластны. Ситуация напоминает фармакологию: как только исследователи изобретают вакцину против конкретного вида гриппа, немедленно появляется новый штамм... Самое увлекательное в нашей работе – это борьба с такими новыми штаммами!»

Оффут отмечает, что конфликт между формальными и неформальными методами реально существует, а не надуман, и что на практике мы не всегда следуем тем принципам, которые провозглашаем (что вызывает сомнение в правильности этих принципов).

Кинг указывает, что это вековечный конфликт, не только не решенный до сих пор, но, возможно, и не имеющий решения. (На самом деле, эта мысль применима едва ли не ко всякой из дихотомий, представленных в данной книге!) Но ясно также, что выбор может зависеть от характера решаемой проблемы: для хорошо изученных задач вполне приемлемыми могут быть формальные методы. Для новых задач (таких как решаемые Оффутом за закрытыми дверями его кабинета!) они, очевидно, непригодны.

Я благодарен обоим этим авторам, а также всем, кто соберется написать мне по данному вопросу (недвусмысленный намек!), за то, что они помогли мне направить мысли в новом и интересном направлении.

Ссылки

SIMON 1965 – *The Shape of Automation for Men and Management*, Harper and Row, 1965; Herbert Simon.

Глава 3 | Оптимизация и разумная достаточность

Введение
3.1. Принцип «лучшее – враг хорошего» и решение задач
3.2. Достаточно хорошее программное обеспечение
3.3. В защиту методов ad hoc
3.4. Михаил Горбачев и продуктивность программирования (!?)

Введение

Неопределенность присутствует в большинстве ситуаций, с которыми нам приходится сталкиваться. Это элемент человеческого существования. Мы должны научиться принимать эту данность и действовать в условиях неопределенности двумя главными способами: находить, если это возможно, средства, чтобы уменьшить ее, либо научиться смиряться с ней... Способность смиряться с неопределенностью имеет решающее значение для творческого мышления и решения задач.

Моше Ф. Рубинштейн

«Tools for Thinking and Problem-Solving»,
Prentice-Hall, 1986

В названии этой главы есть некоторое насилие над английским языком. Оптимизация (optimizing) – вполне литературное слово, производное от optimum – «лучший, наиболее благоприятный».

Другое слово, употребленное в заголовке (satisficing), несколько сомнительно. Оно происходит от совершенно допустимого глагола satisfy – «дать человеку то, в чем он нуждается, чего он требует или хочет; не требовать ничего лишнего, предполагать достаточность». Слово satisficing вы обнаружите не в каждом словаре. В моем его точно нет.

Тем не менее, это совершенно допустимое слово – в определенных кругах. Насколько мне известно, в этих кругах оно и было изобретено.

Что это за круги? Во введении к предыдущей главе я рассказывал о науке решать задачи и принятому в ней мнению, согласно которому эвристики лучше всего подходят для решения сложных задач. Полагаю, именно среди людей, исследующих решение задач, и было изобретено это слово – вероятно, потому что там требовалось решить серьезную проблему.

Проблема состояла в следующем. Пользуясь строгими формальными методами, предположительно можно найти правильное или даже оптимальное решение задачи. Вспомним, что оптимальными мы назвали «лучшие, наиболее благоприятные» решения. Чудесно, когда такие решения находятся.

Однако изучая решение задач, мы сталкиваемся с тем, что с этими несчастными эвристиками никогда не знаешь, действительно ли найденное решение – лучшее. Согласно определению «разумно достаточным» (satisfic-

ing) решением будет «достаточно хорошее» решение, относительно которого трудно утверждать, что оно «лучшее».

Эту дилемму хорошо иллюстрирует рис. 3.1. На данном этапе достаточно будет отметить, что у сложных задач часто бывает несколько решений, и ввиду сложности задач часто невозможно выделить среди них явно лучшее.

Как поступить тому, кто решает задачу? Самые успешные и способные решатели задач отвечают: «Satisfice», – то есть если не можете найти оптимальное решение, найдите то, которое будет разумно достаточным.

Что это означает? Это означает найти решение, которое видится лучшим из всех, что можно получить лучшими из известных вам эвристических способов.

Вы можете сказать, что этого недостаточно, и будете правы. По одну сторону оказываются формальные методы, демонстрирующие свои оптимальные решения, по другую – несчастные эвристики, не дающие ни малейшего представления о том, хороши или плохи предоставляемые ими решения.

Что остается делать решающему задачу? Потребовать, чтобы разумно достаточное решение было *приемлемым* (*supportable*). (В поддержку одного выдуманного слова приходится выдумать другое!). Решение считается разумно достаточным, если оно приемлемо, то есть если имеются веские основания считать, что оно *может* быть оптимальным.

Сложные задачи обладают тем милым свойством, что доказать неоптимальность вашего разумно достаточного решения не так-то просто. Сам факт сложности задачи означает, что едва ли кто-нибудь решится оспаривать ваш «оптимальный» (на самом деле «разумно достаточный») ответ. А если кому-то это удастся, вы просто примете его новое решение в качестве разумно достаточного – маловероятно, чтобы оно действительно оказалось оптимальным, – и подождете, пока появится другое, лучшее, но также разумно достаточное решение!

Каким образом поиск оптимальных и разумно достаточных решений связан с творчеством в программировании? В этой главе я хочу показать, что при разработке задач, наиболее требующих творческого подхода, программисту с большой долей вероятности придется отказаться от надежд на оптимальное решение и, как бы это его ни огорчало, пытаться достичь разумной достаточности. И если такое случится, то творческому программисту нужно понимать значение приемлемости для разумной достаточности.

Под выбранным девизом мы совершим небольшое путешествие. Мы изучим принцип «лучшее – враг хорошего», для которого есть претенциозный акроним из числа тех, что так любимы компьютерщиками (BIEGE –

Best is the Enemy of the Good Enough, и немного поиграем в филологов, рассмотрев ошибочное (по неясным причинам) употребление в компьютерной науке выражения *ad hoc*. Затем мы посетим бывший Советский Союз и попытаемся лучше понять его уважаемого, ныне бывшего руководителя Михаила Горбачева с точки зрения психологической «неоднозначности». Неужели все эти несхожие вещи могут иметь отношение к оптимизации и разумной достаточности?

Мне кажется, да. Лучше разбираясь в понятиях *ad hoc* и «двусмысленность», мы вернее решим, когда от оптимальности нужно переходить к разумной достаточности. Хотя, в лучшем случае, это будет разумно достаточным, а не оптимальным решением. Пусть читатель решает сам!

3.1. Принцип «лучшее – враг хорошего» и решение задач

Сколько труда нужно потратить, чтобы разработать программное решение задачи?

Вопрос звучит достаточно просто, и можно предположить, что ответ на него тоже будет простым. Как в старой загадке про то, какой длины должна быть статья или речь: как юбка – достаточно длинной, чтобы покрывать предмет, и достаточно короткой, чтобы вызывать интерес (есть ли женский эквивалент этой крайне шовинистической мужской шутки?!)

В нашем случае, проект должен быть достаточно проработан, чтобы полностью решать задачу, но не более того. Кажется, все просто?

На самом деле, не совсем. Решения бывают разные. Одни решения могут быть детально проработанными и очень тонкими (их называют «ювелирными»). Другие решения оказываются упрощенными и тривиальными (их называют «неадекватными» или даже «негодными»). Но бывают решения, о которых можно только мечтать, – изящные, красивые и творческие, представляющие собой простое решение невероятно сложной задачи. Хитрость в том, чтобы сначала попытаться достичь такой элегантности, а если этого не позволяют имеющиеся ресурсы, то предоставить сложное решение, сложность которого обусловлена лишь особенностями задачи, а не нашим несовершенством.

Однако во всех этих рассуждениях предполагается, что решить задачу можно несколькими способами, и трудность в том, чтобы определить, достаточно ли хорош тот, который вы рассматриваете в данный момент. Но это как раз и есть краткое описание мира программирования и трудности поиска программных решений. Даже для сравнительно простых задач может не существовать неоспоримо лучшего метода поиска реше-

ния (и неоспоримо лучшего решения), если вы заняты поиском программного решения или даже кодированием решений. Большинство задач в области программирования относится к категории «неструктурированных», а определить объем труда, необходимого для решения задачи этого класса, – ну очень серьезная задача!

(Помню, как на одной из недавних конференций по программированию докладчик предложил механизм поиска компонентов для повторного использования, которому требовалось, чтобы разработчик сформулировал проектное решение в том же виде, в каком это сделал разработчик того самого компонента для повторного использования, который нужно найти. Такие совпадения в реальном мире программных решений просто исключены.)

В действительности, для большинства сложных задач есть много различных уровней проектных решений. Есть идеальное, лучшее из всех возможных решение. Есть приемлемое решение – «работает, но хотелось бы чего-то лучшего». Есть совершенно неадекватное решение – «это никуда не годится». Плюс множество промежуточных ступеней. Рисунок 3.1 прекрас-

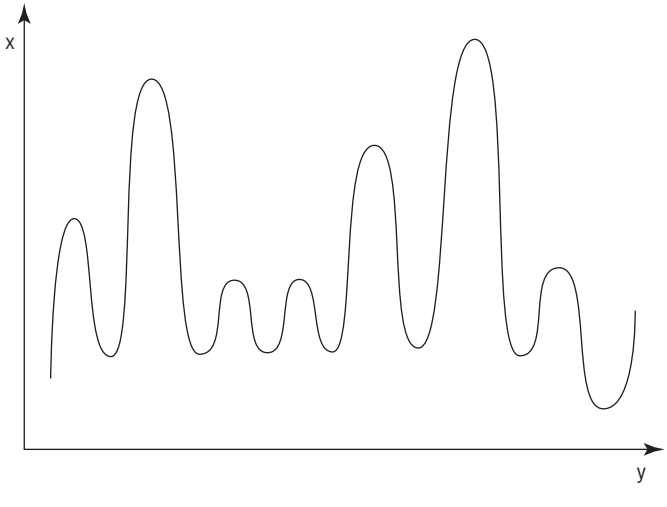

Рис. 3.1. *Оптимальность и разумная достаточность. Предлагается метод поиска максимальной y-координаты этой кривой вслепую методом проб и ошибок. (Считаем, что аналитических методов нет. В данном случае такое допущение может быть оправданным.) Если не сделать шаг поиска достаточно маленьким (это было бы дорого), то найденное решение окажется, скорее всего, не оптимальным*

но иллюстрирует подобное многообразие решений для задачи поиска экстремума изображенной кривой.

Теперь вся проблема осложняется. Если различных уровней решений много, то снова возникает вопрос о том, сколько труда требует поиск решения.

К счастью, некоторые ответы можно найти в литературе, посвященной решению задач. Любопытно, что методами решения задач люди, надо полагать, интересовались испокон веков, но лет тридцать назад, с появлением такого инструмента решения задач, как компьютер, интерес резко возрос. И одна книга, посвященная решению задач, по моему убеждению, возвышается над прочими.

Это «Науки об искусственном», шедевр Герберта Саймона, лауреата Нобелевской премии и премии Тьюринга. Объем этой книги таков, что данный скромный очерк выглядит в сравнении с ней карликом, и самое увлекательное в ней – понятия «оптимизации» и «разумной достаточности». Оптимизация – это попытка найти лучшее из имеющихся решений задачи. Однако Саймон утверждает, что для сложных задач часто нет очевидного способа поиска оптимального решения. Для такого класса задач нам необходимо, по мнению Саймона, отправиться на поиски разумно достаточного решения.

Что он понимает под «разумно достаточным»? Он говорит о таком решении, которое работает, причем «достаточно хорошо», но не дотягивает до оптимального, хотя и выполняет свою функцию.

Как установить, что ваше решение именно такое? Согласно Саймону для этого вы должны представить адекватное обоснование своего решения. Таким образом, разумно достаточное решение должно быть «приемлемым» (supportable).

Может быть, это всего лишь семантическое словоблудие? Что ж, вернемся к кривой на рис. 3.1 и попробуем предложить метод, с помощью которого можно гарантировать нахождение истинного максимума. Взглянем на дело с практической точки зрения. Ваши ресурсы для поиска решения небезграничны. Может быть, ваш начальник, как обычно, напоминает, что решение должно было быть найдено еще неделю назад, а может, он жестко ограничил ваши возможные затраты на поиск решения.

Понятие «разумной достаточности» представляется весьма важным, но оно дополнительно осложняет процесс решения задач. Теперь мало начать искать решение – нужно заранее установить, какой уровень решения будет приемлемым. Может быть, вы столкнулись с той редкой задачей, которой не подходит никакое решение, кроме оптимального? Или у вас ограничения по времени и стоимости, и вам нужно просто работо-

способное решение? В мире встречаются задачи обоих типов, и важно понять свою ситуацию, прежде чем начать поиск решения.

Здесь весьма уместна одна чудесная поговорка. Это принцип «лучшее – враг хорошего».

Из него попросту следует, что если для вашей задачи подходит разумно достаточное решение, то не стоит тратить силы на поиск оптимального.

3.2. Достаточно хорошее программное обеспечение

Все вышесказанное носило довольно философский характер. Теоретически, я думаю, почти все с этим согласятся. Разумеется, «достаточно хорошее» решение можно улучшить, потратив дополнительное время на его мелкие усовершенствования, но только закоренелый перфекционист сочтет нужным улучшать «достаточно хорошее».

Но лет десять назад эти вопросы перестали быть философскими и стали достаточно конкретными. Сторонники принципов «разумной достаточности» и «лучшее – враг хорошего» начали пропагандировать то, что они назвали «достаточно хорошим ПО». Хотя в свое время масса сил была потрачена на определение «достаточно хорошего», в контексте наших рассуждений в нем, как мне кажется, нет нужды. Договоримся считать, что «достаточно хорошее» = «разумно достаточное».

Наверное, неудивительно, что многие заняли жесткую позицию по этому вопросу. Одни заявляли, что нет смысла стремиться к совершенству в таком деле, где достаточно трудно просто заставить что-либо работать. Другие, наоборот, видели опасность в том, что якобы снижаются стандарты в профессии программиста и поощряется написание низкокачественных программ, и это в такие времена, когда на рынке вообще трудно найти удовлетворительно работающую программу. Трудно было надеяться на смягчение разногласий.

И пока все это происходило, издания вроде «Cutter IT Journal» подливали масла в огонь, посвящая данной теме специальные выпуски и стимулируя публичные дискуссии. Честно говоря, мне кажется, что «IT Journal» поступал в этом случае правильно. Такие темы, если не откладывать их разбор, имеют тенденцию незаметно тлеть и ярко вспыхивать в самый неожиданный момент.

В каком-то роде все это, конечно, свелось к буре в стакане. Одни действительно пытались определить методологии и технику, которые позволили бы делать «достаточно хорошие» программы. Другие продолжали бороться с этими технологиями по изложенным выше мотивам. Но факт то, что

по большей части результаты нашего труда оказываются «достаточно хорошими», нравится нам это или нет, потому что добиваться чего-то большего обременительно как в финансовом, так и в умственном отношении. Подход «лучшее – враг хорошего» вполне прижился в мире программирования, и критика принципа «достаточно хорошего» стихла сама собой.

Чтобы этот спор не казался вам слишком философским, приведу пример, на мой взгляд, неплохо иллюстрирующий проблему. В некие времена одному блестящему специалисту по компьютерным наукам, работавшему в аэрокосмической фирме, предложили написать прикладную программу. Требовалось создать ассемблер для одноразового бортового компьютера, устанавливаемого на космическом корабле. Все приложение в целом было, конечно, уникальным, как и большинство приложений в космонавтике, в отличие от данной конкретной задачи. Всякий, кто разбирается в системном программировании, может написать ассемблер, и с этой задачей можно было справиться достаточно быстро.

Но не в характере этого блестящего специалиста было делать что-либо на скорую руку. Работающую версию ассемблера он изготовил весьма скоро, что неудивительно. Но то, что за этим последовало, поразило всех остальных участников этого проекта.

Своего часа ожидали другие программные задачи, для которых не были еще назначены исполнители. Но наш блестящий специалист все работал над своим ассемблером, постепенно превращая его в произведение искусства. Он усовершенствовал интерфейс пользователя; он разукрасил вывод данных; он осуществил рефакторинг кода, сделав решение более элегантным.

Шли недели. Все остальные участники завершили свои задачи и перешли к следующим. Но совершенствование ассемблера все продолжалось. Разумеется, без всякой в том необходимости. Но результат становился все лучше.

Впервые услышав поговорку «лучшее – враг хорошего», я уже знал, чем это оборачивается на деле, поскольку вполне усвоил уроки того конкретного случая. Правда, тогда мы называли такую ситуацию иначе – «наведение лоска» (goldplating). И начальники зорко следили за тем, чтобы их подчиненные этим не занимались. (Думаю, начальник того блестящего компьютерного специалиста просто не понимал, что для создания ассемблера не нужно столько времени. Вспомните островолосого босса из комиксов Дилберта!)

И когда заходит речь о принципе «лучшее – враг хорошего», я мысленно переношусь в те времена, когда последствия его нарушения были весьма печальными. С тех пор я твердо встал на сторону тех, кто поддерживает идею «достаточно хороших» программ.

3.3. В защиту методов ad hoc

Какие слова программист-теоретик произносит как самое грязное ругательство?

По моим наблюдениям последних лет – это *ad hoc*. Так специалисты по компьютерным наукам называют какую-нибудь порочную практику написания программ. Это может означать «нестрого», «не планомерно» и даже «хаотически». Если какой-нибудь теоретик охарактеризовал ваш стиль работы как *ad hoc*, знайте, что он невысокого мнения о ваших способностях.

Признаюсь кое в чем. Когда-то впервые услышав, как теоретики что-то клеймят словами *ad hoc*, я не понял, что они имели в виду. Поэтому не поленился заглянуть в словарь и в том, что увидел там, не нашел ничего дурного. В то время я нашел в словаре такое объяснение:

> *ad hoc:* созданный для специальной цели, например, «специальная комиссия» («Оксфордский американский словарь»).

Я был озадачен. В словаре не было ничего такого, что объясняло бы всю ненависть, которую теоретики вкладывали в выражение *ad hoc*. Если я всего лишь делаю что-то со специальной целью, это вовсе не значит, что я делаю это плохо. В чем же дело?

Шло время. Загадка не выходила у меня из головы, ютясь где-то на задворках памяти. Теоретики продолжали применять это выражение в каком-то осуждающем смысле. Пожалуй, выражение *ad hoc* еще сильнее стало ассоциироваться с чем-то дурным. Даже в правительственных докладах о перспективах развития программного обеспечения появились высказывания о том, что мы должны перейти от старых хаотических и *ad hoc*-методов работы к неким новым и, разумеется, более прогрессивным.

Недавно мне попалось некое библиотечное чудо – словарь такой толщины, что поднять его можно было только краном! «Ага, – подумал я, порывшись у себя в памяти, – может быть, этот словарь и поможет мне разрешить загадку выражения *ad hoc*». Пыхтя от натуги я раскрыл этот огромный словарь и снова отыскал определение *ad hoc*:

> *ad hoc* – для данной специальной цели; по отношению к данной теме или предмету («Вебстерский иллюстрированный энциклопедический словарь»).

Загадка не только не разрешилась, но оказалась еще сложнее. Мне стало ясно, что ученые теоретики неверно употребляют это выражение!

Закрыв этот второй словарь и отдышавшись (!), я вернулся к размышлениям о данной семантической странности. Почему целое поколение спе-

циалистов по компьютерным наукам бездумно, как лемминги, друг за другом повторяло эти слова в неверном смысле?

Я пришел к выводу, что здесь сыграли роль два обстоятельства. Первое относится к истории вычислительной техники, а второе – к происхождению компьютерных наук. Обратимся сначала к первому.

С самого момента появления в начале 1950-х коммерческой вычислительной техники некие таинственные и замечательные люди уже думали над тем, как создавать то, что мы называем программным обеспечением. На первых порах, несмотря на все трудности, удавалось сосредоточиться на конкретной решаемой задаче, придумать для нее решение и написать соответствующий код. Обычно он писался в машинных командах или на ассемблере, выполнялся на относительно пустой машине, где операционная система если и присутствовала, то в зачаточном состоянии, и решал задачу эффективным (иначе в те времена не допускалось) и простым (для сложности у тех машин было недостаточно ресурсов) способом.

Проблема заключалась в том, что для каждой новой задачи пользы от каких-либо ранее реализованных решений было не много. Нужно было как-то обобщить процесс программного решения задач. Для выработки обобщенных методов решения прилагалось все больше усилиий. Были изобретены универсальные языки программирования, появились универсальные вычислительные машины, универсальные инструменты и компоненты, были заложены основы единообразных процессов, применявшихся в разных проектах.

В те времена большинство программистов с удовольствием переходили на новые технологии. В конце концов, тогда, чтобы заниматься программированием, нужно было постоянно осваивать новое, и технологии менялись так быстро – каждые пару лет просто кардинально, – что если вы не меняли свои методы, то почти мгновенно оказывались на обочине. Чтобы не отстать от времени, явно требовался переход от специфических методов к универсальным.

Вернемся к псевдоругательству *ad hoc*. В те времена, если вы занимались только особыми решениями конкретных проблем и не стремились двигаться вровень со стремительно развивавшимися универсальными технологиями, то явно были «плохим» программистом. Я хочу сказать, что был период, когда имело смысл относить *ad hoc* ко всем худшим способам создания программ.

А как насчет второй причины неправильного употребления *ad hoc* – той, что связана с зарождением компьютерных наук? Многие программы изучения компьютерных наук имеют математическое происхождение, и многие ученые в области компьютерных наук по-прежнему сильно тяготеют

к этому наследию. Здесь есть как плюсы, так и минусы, и жаль, что математики раскололись на два лагеря: в одном возвеличивают «чистую» математику, в другом – «прикладную». Те и другие смотрят друг на друга свысока. На математических факультетах между ними идет такая борьба за влияние, что не поверишь, если сам не видел.

Как это наследие влияет на компьютерные науки? Рискну утверждать, что большинство специалистов по компьютерным наукам ближе к лагерю чистой математики, чем прикладной. Это означает, что они гораздо охотнее решают интеллектуальные задачи, связанные с компьютерами, а не с прикладной областью. Попавшие в эту западню склонны свысока смотреть на все, что связано с практикой/конкретикой. И все, что делается *ad hoc*, то есть для решения конкретной прикладной задачи, им не интересно. Более того, при такой узкой точке зрения все, кто применяет специфический подход для решения конкретной задачи, воспринимаются как действующие неадекватно, хаотически.

Вот какого многословия потребовало определение одного-единственного термина. Есть ли в этом смысл? По-моему, безусловно есть.

Я считаю, что за прошедшие годы в компьютерных науках проделана неплохая работа по разработке и пропаганде универсальных решений задач, которыми занимаются программисты. Представления, приемы, методы и инструменты, явившиеся результатом этого подхода, в целом замечательны. Однако я полагаю, что эти компьютерные специалисты практически достигли предела того, что могут дать универсальные методы. Убежден, что для развития компьютерных наук необходимо учитывать прикладную сторону.

И мы уже видим начало движения в этом направлении. «Анализ предметной области» (domain analysis) – термин, которым компьютерные специалисты обозначают исследование возможностей повторного использования в прикладной области. Появились языки программирования, ориентированные на конкретное применение, методы, действующие лучше в одних областях, чем в других, и даже делаются попытки создания специализированных аппаратных платформ, таких как машина базы данных.

Кроме того, на протяжении многих лет программисты занимаются решениями, которые просто не могут не быть *ad hoc,* будучи нацеленными на конкретную проблему:

- Системный анализ, где аналитики определяют задачи, которые нужно решить.
- Нисходящее проектирование, нацеленное на решение задачи соответственно предъявленным техническим требованиям.

- Нисходящая реализация, имеющая целью создать скелет общего решения задачи для последующего присоединения к нему блоков более низкого уровня.
- Разного рода тестирование. Например, при «тестировании граничных значений» создаются контрольные примеры для проверки граничных ситуаций, существующих в конкретной области, а при «статистическом тестировании», являющемся частью методологии Cleanroom, контрольные примеры создаются на основании «характера применения», свойственного рассматриваемой задаче.

По моим наблюдениям, применение подобных *ad hoc*-подходов расширяется. И я считаю, что это должно приветствоваться, а не вызывать тревогу.

Интересно, что в литературе, посвященной решению задач, принято различать «сильные методы», предназначенные для решения конкретной задачи, и «слабые», более универсальные по своей природе. Сильный метод можно сравнить с гаечным ключом заданного размера, который оптимально приспособлен для выполнения одной конкретной работы. Слабый метод похож на разводной ключ, который можно приспособить для решения множества задач, но ни для одной из них он не будет оптимальным.

Ошибочное употребление в компьютерных науках термина *ad hoc* слишком долго уводило нас от сильных методов в сторону слабых. Пора к ним вернуться. Для начала нужно освободить вполне достойный термин *ad hoc* от лишней нагрузки.

3.4. Михаил Горбачев и продуктивность программирования (!?)

Когда-то мне попался текст, не имеющий, казалось бы, никакого отношения к эффективности разработки программного обеспечения и, тем не менее, содержащий мысли, которыми мне хочется поделиться с читателем.

Речь идет о заметке «Psyching Out Gorbachev» (Психологический портрет Горбачева), опубликованной в рубрике «Insight» газеты «Washington Post» 17 декабря 1989 года. В ней личность Михаила Горбачева анализировалась тремя крупными психологами, изучавшими политических лидеров и их мысли (на основании их устных и письменных высказываний). Один из этих психологов, Филип Тетлок (Philip E. Tetlock), разработал систему оценки сложности мышления и способности действовать в условиях неопределенности, свойственных, по его мнению, крупным лидерам.

По семибалльной шкале Тетлока Горбачев получил оценку, близкую к максимальной. (Некоторых американских политиков, в частности, се-

натора Джесси Хелмса, он оценил одним баллом.) Чтобы получить семь баллов, нужно уметь видеть альтернативные подходы, взвешенно оценивать их и либо действовать в направлении разрешения противоречия, либо терпимо относиться к неопределенности положения, если нет достаточных оснований для принятия мер. Тетлок отметил любопытный факт: тех, кто в сложных и неопределенных условиях ведет себя спокойно, часто считают слабыми (именно так, по его словам, в России воспринимали Горбачева), и наоборот, тех, кто склонен к большей непримиримости, считают сильными. Мнение самого Тетлока по этому поводу противоположно.

Как все это связано с творчеством в программировании?

Я исхожу из двух тезисов. Во-первых, в такой сложной и новой сфере деятельности, как компьютерные вычисления и обработка данных, накоплено слишком мало знаний, а потому мы занимаемся своим делом в условиях высокой неопределенности. Мы действительно не знаем, какие способы написания программ лучше, и вряд ли выясним это в ближайшем будущем.

Во-вторых, степень нашей неосведомленности такова, что нам следует возможно более открыто, честно и терпимо высказывать свои мнения и изучать разногласия.

Например, я считаю, что по общим вопросам эффективности разработки программного обеспечения у нас меньше знаний и ясности, чем по любой другой проблеме в области программирования. Нашей профессии лишь немногим больше полувека. Можно ли было ожидать иного состояния этой отрасли при таком сроке?

И вот, в условиях недостаточности знаний и неопределенности появляются два идеологических течения. Одно из них провозглашает каждую новую возникшую идею окончательным решением проблемы продуктивности в разработке программного обеспечения. Сначала это были структурные методы, потом языки четвертого поколения, затем CASE-средства, за которыми последовало объектно-ориентированное программирование. Те, кто принадлежит к этому течению, являются, по моему мнению, первоклассными теоретиками. Кому-то они кажутся сильными благодаря способности ясно видеть решение и быстро двигаться в выбранном направлении. Другим они представляются примитивными в силу склонности упрощать сложные проблемы и неспособности существовать в условиях неопределенности.

С другой стороны, есть люди вроде Фреда Брукса с его статьей «No Silver Bullet» (Серебряной пули нет) и Дэвида Парнаса с его статьями «Star Wars» (Звездные войны) – они не предвидят никаких прорывов в обозри-

мом будущем и склонны высказываться в духе «создание программ – по самой своей природе сложная задача».

Вернемся снова к Тетлоку и Горбачеву. По словам Тетлока, у политиков редко встречается терпимое отношение к сложным ситуациям. «Обычно, – пишет он, – они гораздо больше склонны к простым и жестким дихотомиям».

Должен отметить, что терпимое отношение к сложным ситуациям столь же нехарактерно для руководителей, предпринимателей и научных работников в области программирования. Простые решения и образ сильного деятеля для многих оказываются слишком соблазнительными.

Это наносит ущерб программированию – области, в которой мы работаем. Когда очередное предложение по увеличению продуктивности вместо обещанного прорыва приносит весьма скромные результаты, мы быстро хватаемся за новое. Мы как будто все время повторяем: «Где-то там нам непременно повезет, нужно только в это верить».

Возможно, так и будет. Но перевороты нельзя планировать, они случаются, когда их меньше всего ждут. Вера и громкие заявления их не приближают.

Для тех, кто в программировании не только решает текущие задачи, но и стремится к переменам, изучение личности Михаила Горбачева – возможно, самого выдающегося реформатора нашего времени – отнюдь не бесплодное занятие.

Сложность и неоднозначность. Несомненно, они свойственны многим аспектам программирования. Пора принять их как данность, не стремясь изжить любой ценой.

Где те семибалльные лидеры в нашем деле, которым хватит смелости занять эту позицию?

Глава 4 | Количественный и качественный подходы

Введение
4.1. «Нельзя управлять тем, что нельзя измерить» – так ли это?
4.2. Математика и компьютерные науки
4.3. Роль интуиции в принятии решений
4.4. Тьма ловушек: числа бывают разные

Введение

Одна из прочнейших традиций в науке – это почтительное отношение ко всему количественному, точному, строгому и безусловно истинному. Однако фактически в окружающем мире господствуют неточность и неопределенность, и безоговорочно утверждать что-либо о нем трудно. Фактом является и то, что точность и определенность обходятся дорого. В стремлении соблюсти подобное «почтение» мы склонны закрывать глаза на эти факты, упуская из виду величину издержек, связанных с достижением высокой точности и малой неопределенности. Другой аспект нашего стремления к почтительности – то, что в большей части научной литературы элегантность берет верх над насущной важностью.

Лотфи А. Заде

«Soft Computing and Fuzzy Logic»,
IEEE Software, ноябрь 1994, ©1994 IEEE

Раздел «Q» словаря невелик, и почти все слова там начинаются с «qu». Но на этих нескольких страницах выделяются две интеллектуальные вершины – слова quantitative (количественный) и qualitative (качественный).

«Количественный» относится к числам: «количество или число предметов; возможность измерения благодаря наличию размера, веса, количества или численности».

С другой стороны, «качественный» означает «имеющий качество или стремящийся к нему», где качество определено как «степень или уровень совершенства».

Размышляя над этими двумя определениями, я ощутил некоторую нерешительность. В обществе принято строго различать количественные и качественные методы. Одни вещи допускают численное описание, другие – нет. Исходя из этой традиции, я полагал, что данные два слова ясно и четко различаются между собой.

Но, постойте! Если понятия *качества* и *количества* столь различны, значит ли это, что о качестве нельзя говорить на языке чисел? Но если качество можно измерить количественно, то между этими двумя понятиями нет непреодолимой преграды.

Конечно, понятие качества весьма смутно. Оно завело в умственный тупик главного героя книги Пирсига «Дзен и искусство ухода за мотоциклом». Думаю, оно вызывает больше всего раздоров среди любителей давать определения в компьютерной области. (Чтобы проследить за ходом их мыслей, посмотрите обсуждение моей книги «Создание качественного программного обеспечения» («Building Quality Software», Prentice-Hall, 1992).) Если так трудно дать определение этому понятию и люди сходят с ума, просто пытаясь выяснить его смысл, удастся ли подойти к нему с количественной точки зрения?

Ответом будет «мы все время это делаем» или, по крайней мере, «все время пытаемся это делать». Мы оцениваем качество бегунов, измеряя скорость их бега. Мы оцениваем качество прыгунов в длину, измеряя расстояние, на которое они прыгают. И если нет другого выхода, мы оцениваем качество ныряльщиков и конькобежцев с помощью коллегий судей, которые показывают карточки с цифрами своих оценок за выступление. Если нет реальных количественных оценок для качества, можно их придумать.

В программировании мы тоже постоянно пытаемся измерять качество. Мы определяем количество ошибок, скажем, на строку кода или функцию, говоря при этом, что измеряем надежность. Мы подсчитываем операторы и операнды (и массу других вещей), говоря при этом, что измеряем сложность. Мы хронометрируем прогон программы, говоря при этом, что измеряем ее производительность. Мы подсчитываем инструкции и данные, говоря при этом, что измеряем эффективность использования памяти. Все, что мы здесь измеряем, относится к неуловимой вещи под названием качество.

Но не станем слишком далеко заходить в эту сторону. Несмотря на все упомянутые здесь количественные аспекты качественных оценок, между этими двумя понятиями все же существует фундаментальная разница. Одни вещи легко представить на языке чисел. Другие – нет. Это фундаментальное различие, и его значение в области программирования я собираюсь обсудить в этой главе.

Что общего между количественными методами, качественными методами и творчеством? Мое мнение таково: углубляясь в область гибких, эвристических, разумно достаточных методов решения задач, то есть тех аспектов творчества в программировании, которым посвящены предыдущие главы, мы переходим из той сферы, где легко делать измерения, в другую, где это не так просто. Мечтать о том, чтобы задачи этого мира можно было решать, в надлежащей мере применяя дисциплину, формальные методы и оптимизацию, – все равно что желать, чтобы все в этом мире легко допускало измерение.

Вопреки нашим желаниям мир, в котором мы живем, – мир сложной культуры, интеллекта и творчества – оказывается не так прост. Сравните, например, эту книгу с другими статьями о программировании, которые вы, возможно, прочли. Как бы вы стали сравнивать ее с последними книгами Вейнберга, или Юрдона, или Демарко? Чего бы все мы хотели – в том числе я, как бы ни разочаровал меня результат, – так это дать какую-нибудь ясную количественную оценку каждой книге, чтобы можно было все их расставить по порядку. Скажем, Демарко получает 9,723 балла из 10 за свой светлый ум, нестандартные взгляды и юмор. Юрдон – 9,683 за исчерпывающее освещение, огромные связи и широчайший круг источников информации. Вейнберг – 9,706 за изобретательное сочетание психологических и технических идей, описание удивительных случаев и невероятную продуктивность. Гласс? Ему 1,738 балла за мучения.

Однако все это достаточно проблематично (даже если забыть о том, что я занял последнее место!). Немногие отнеслись бы к подобным числовым показателям с доверием. Во-первых, они субъективны. (Рискну предположить, что, к примеру, моя мама дала бы авторам следующие оценки: Гласс – 9,999, Демарко – 1,111, Юрдон – 1,111, Вейнберг – 1,111). Во-вторых, трудно оправдать значение составляющих эти оценки чисел, особенно по мере удаления вправо от запятой. Иными словами, мы ввели количественную оценку там, где она ни к чему. Качественный подход был бы здесь более уместен. Таки есть действительная разница между этими двумя подходами!

Чем же мы займемся в этой главе? Сначала мы рассмотрим один из вредных компьютерных трюизмов: «Нельзя управлять тем, что нельзя измерить». Как и большинство бабушкиных (и дедушкиных) сказок, он гораздо менее правдив, чем может показаться на первый взгляд.

Затем мы рассмотрим еще несколько примеров связи количественных методов с компьютерной обработкой, каждый раз убеждаясь, что качественные методы, хотя и менее привлекательные, иногда оказываются единственно доступными. Наконец, подобно столяру, с удовольствием забивающему в изделие последний гвоздь, мы закрепим свою точку зрения последним разделом. Интересно, как вы оцените эту главу по шкале от 1 до 10 баллов?

4.1. «Нельзя управлять тем, что нельзя измерить» – так ли это?

«Нельзя управлять тем, что нельзя измерить» – один из известных компьютерных и программистских трюизмов.

Но справедлив ли он? Правда ли невозможно управлять тем, что нельзя измерить?

Отвечаю: неправда. На самом деле, мы занимаемся этим постоянно. Приведу несколько достаточно общих примеров.

Начнем с того, что мы уже несколько десятилетий управляем программными проектами, обычно не пользуясь при этом никакими метриками. Конечно, можно говорить о том, что мы делаем это довольно плохо, но несомненно, что даже при «плохом» управлении мы успешно вступили в «компьютерную эпоху», которая войдет в историю как одно из самых важных предприятий человечества. Могло ли это произойти, если бы мы действительно не управляли тем, что не можем измерить?

Мы управляем исследованиями и разработками – деятельностью, фактически не поддающейся измерению, и результаты этой работы изменили облик мира.

Мы управляем руководителями. В иерархически организованной сфере корпоративного управления, чем более высокую должность вы занимаете, тем труднее количественно оценивать работу ваших подчиненных. И даже если вы будете оспаривать успешность наших действий, мы все же делаем это.

Фактически, самый общий из возможных здесь выводов состоит в том, что, управляя работниками информационной сферы, вы, весьма вероятно, окажетесь в области, где роль количественных методов невелика.

Поэтому, думаю, справедливо будет сказать, что указанный трюизм не верен. Мы на самом деле управляем тем, чего не можем измерить, и именно сейчас, когда вы читаете эти строки, этим занимаются тысячи руководителей по всему свету.

Конечно, хотелось бы управлять на количественной основе, имея для этого количественные показатели. Однако и в отсутствие соответствующих метрик мы справляемся с управлением.

Тут нам, несомненно, повезло. Управление программными разработками десятки лет существует без надежных численных показателей, на которые можно опереться. Программные продукты, благодаря которым наступила компьютерная эпоха, пришли из той области, где не было ни надежных показателей, ни проверенных рекомендаций о том, как их применять.

Конечно, попытки были. Мой начальник в конце 1950-х пытался реализовать достаточно мощные по тем временам методы измерений для оценки разработки программного обеспечения. Мне было поручено определять и внедрять некоторые метрики. Насколько я припоминаю, они были достаточно детальны и конкретны и далеко не ограничивались привычным заданием контрольных точек, включая измерение таких вещей, как степень подробности комментариев, которыми снабжались все элементы данных в программе, а также единиц измерения этих элементов (думаю, можно назвать их «метриками метрик»).

Это была одна из первых интересных попыток управлять с помощью количественных показателей, но она оказалась мертворожденной. Большинство коллег этого руководителя не проявили интереса к управлению на таком уровне детализации, и никто больше не применил эти методы. Технические работники не приветствовали то, что казалось им вмешательством руководства в их работу, и всячески уклонялись от нововведений. Этот руководитель ушел на повышение, и вводимые им метрические подходы без него зачахли.

Лет десять спустя подобные опыты появились и в научных кругах. Мюррей Холстед (Murray Halstead), известный своей работой над созданием компиляторов, сделал примечательную попытку внедрить науку и измерения в программирование, предложив нечто под названием «научное программирование» (software science). Интересно отметить, что примерно тогда же появились термины «программная инженерия» (software engineering) и «программная физика» (software physics). Некоторые из этих понятий сохранились до нашего времени. «Научное программирование», как известно сейчас многим, занималось оценкой сложности программ путем подсчета количества операторов и операндов.

Работа Холстеда и его последователей вызвала огромные споры. Допустимость предлагаемых оценок критиковалась как учеными, так и практиками, причем ученые сомневались в правильности рассуждений Холстеда, а практики – в пользе от его изысканий. Страсти разгорались при одном лишь упоминании предмета. На одной из компьютерных конференций при обсуждении научного программирования один видных деятелей программной инженерии обвинил другого в приверженности «астрологии» (имея в виду лженауку). В настоящее время большинство авторитетов в этой области выражает сомнение в ценности этой идеи, но, тем не менее, количество посвященных ей публикаций в научной печати огромно, а также, что любопытно, появился ряд довольно успешных коммерческих продуктов, занимающихся расчетом этих метрик.

Те, кто оценивает практическое состояние дел, находят, что применение метрик в управлении программными проектами еще не вышло даже из

пеленок. На конференциях «Применение метрик в программировании» («Applications of Software Metrics») Билл Хетцел (Bill Hetzel), представляющий организацию-спонсора, регулярно докладывает о распространенности метрик на практике и считает ее ужасающей.

Любопытно, однако, что недостатка кандидатов в метрики нет. В действительности, в области метрик наблюдается раздвоение. Метрики, предлагаемые учеными, включая Холстеда и его многочисленных последователей (Зусе (Zuse) составил едва ли не энциклопедию по этой теме), развивались в одном направлении. Метрики же, предлагаемые практиками и якобы не используемые, довольно успешно развивались, но в другом направлении (для которого типичны работы Роберта Грейди (Robert Grady) из Hewlett-Packard).

(Разумеется, слово «успешно» в предыдущем абзаце употреблено в ироническом смысле. Как можно считать что-то успешным, если оно малоупотребительно? Однако справедливо будет заметить, что если сегодня какая-либо фирма-производитель программного обеспечения пожелает внедрить у себя программу применения метрик, она найдет богатый выбор как практических, так и теоретических предложений.)

Проблема тезиса «нельзя управлять тем, что нельзя измерить» связана с вызываемым им фанатизмом. Те, кто глубоко уверовал в полезность метрик, огорчаются отсутствием какого-либо прогресса в этой области. Чтобы привлечь внимание спонсоров и исполнителей, они придумывают броские фразы. Но такие фразы, прежде чем поверить им, следует хорошенько анализировать.

Можно ли управлять тем, что нельзя измерить? Конечно, да.

Если нам будет дан выбор, предпочтем ли мы методы управления, основанные на измерениях? И снова ответ утвердительный.

Мир гораздо сложнее, чем это следует из крылатых фраз, которыми мы его описываем. И вздорность обманчивых фраз вроде «нельзя управлять тем, что нельзя измерить», отнюдь не стимулирующих поиски, отчасти объясняет нынешнее состояние дел, то есть отсутствие каких-либо измерений.

4.2. Математика и компьютерные науки

Компьютерные науки выросли на факультетах математики. Там они делали первые шаги, там достигли известности и остаются в тесной связи с математикой. Во многих колледжах и университетах математика и компьютерные науки сосуществуют в рамках единой организационной структуры.

Но насколько эта связь с математикой важна для компьютерных наук? В особенности в том, что касается программирования?

Многие сочтут такой вопрос еретическим. «Как можно сомневаться? – скажут они. – Логика и точность математического представления – неотъемлемая часть компьютерных наук, так было всегда и будет впредь». И это трудно оспорить.

Однако есть и другая точка зрения. Она более сложна и необычна, и потому требует тщательной формулировки.

Когда-то компьютеры служили только для вычислений. Фактически, у них тогда и не было возможностей заниматься чем-то другим. Никаких других символов, кроме цифр, они не знали, и невозможно было представить, что эти удивительные новые устройства годятся для чего-то еще.

То была эпоха вычислений как раздела прикладной математики. Компьютеры выполняли с числами немыслимые трюки, которые прежде считались невозможными или требовали невероятного количества математиков и чудовищно долгого времени. Поначалу компьютеры могли повысить производительность труда только тогдашних математиков.

Но эта эпоха закончилась 50 лет назад. Как только набор символов, которыми оперировали компьютеры, вышел за пределы десяти цифр, они превратились, я бы сказал, в «информационные процессоры». Прошли те дни, когда шестибитный байт позволял писать только заглавными буквами, и сегодня, когда мы стремимся заложить в «вычислительную машину» все символы всех существующих в мире языков, основной материал для компьютера – уже не числа, а оцифрованная информация.

Возникли другие научные дисциплины, в которых вычисления и программирование рассматриваются с точки зрения информации, а не математики. Однако, так или иначе, компьютерные науки остаются самым влиятельным участником в этой области, а они продолжают рассматривать математику как сердцевину всего. Например, в правительственных докладах, посвященных будущему программирования, часто звучат призывы «добавить математики» во все еще незрелые теорию и практику программирования.

Между тем, за прошедшие десятилетия характер компьютерной практики изменился. Вначале вычисления, ориентированные на математику и информацию, представляли собой две весьма различные и слабо связанные дисциплины. Специалисты по вычислениям были воспитаны на механических калькуляторах и рассматривали компьютеры как чрезвычайно быстрые и надежные арифмометры. Специалисты по информации выросли на перфокартах и рассматривали компьютеры как чрезвычайно быстрый и надежный инструмент для обработки перфокарт. Математиче-

ские компьютеры разговаривали на языке двоичных слов, состоящих из фиксированного количества бит, и поначалу редко использовали какие-либо «символы», даже цифры. Напротив, информационные процессоры пользовались цифрами и символами, а «слово» рассматривалось динамически: программы устанавливали «метку слова», чтобы указать, сколько символов содержит текущий вариант отдельного информационного поля.

Постепенно различия в аппаратном обеспечении для нужд вычислений и обработки информации стерлись. В начале 1960-х было замечено, что двоичные компьютеры с возможностью обработки символов, а также десятичных и двоичных вычислений оказываются быстрее и дешевле, чем прежние машины для обработки информации, ориентированные на символы и цифры. Когда в середине–конце 1960-х на сцене появилась IBM 360, аппаратные различия между этими двумя областями фактически исчезли. Одни и те же компьютеры могли экономично выполнять обе функции, и необходимость разделения аппаратных платформ отпала. Да, в информационном процессоре по-прежнему десятичные вычисления выполнялись как дополнение к двоичным, но это было незначительное отличие по сравнению с остальными частями компьютера, большинство которых занималось более существенными вещами, чем просто арифметика.

Современные компьютеры, и это придется признать даже самым упорным приверженцам математики, применяются гораздо шире, чем только для арифметики и математики. Существуют текстовые процессоры, тренажеры, игры и различные инструменты программирования, такие как компиляторы, в которых практически нет математики. Конечно, есть и электронные табличные процессоры, выполняющие значительный объем расчетов, а также научные и инженерные программы, где математика существенна, но все математические приложения, безусловно, занимают меньшую часть всего спектра практического применения компьютеров.

Закончив экскурс в историю, вернемся к нашему исходному вопросу. Насколько важна для компьютерных наук связь с математикой? Как следует из нашего краткого исторического обзора, значение математики как одной из функций сильно понизилось.

Но, может быть, она сохраняет свое скрытое значение? То есть, существуют математические идеи и мастерство, которыми важно владеть компьютерному специалисту, даже если ему не приходится применять математические методы? Многие компьютерные специалисты придерживаются такой точки зрения.

Любопытно, что при ответе на этот вопрос возникает дихотомия между аппаратным и программным обеспечением. Специалисту по аппаратной части ответ явно кажется положительным. Но специалисты по компью-

терным наукам простирают эту важность и на сферу программного обеспечения.

Я называю это «синдромом латыни». Давным-давно, когда в школах постепенно переставали преподавать латынь, кто-то во всеуслышанье заявлял, что умственные навыки, приобретаемые при изучении латыни, имеют большое значение для освоения других учебных дисциплин. Благодаря именно такому ходу мысли латынь просуществовала в качестве учебного предмета гораздо дольше, чем требовало ее практическое значение. Думаю, математика может оказаться для компьютерных наук тем же, чем в свое время стала латынь для прочих образовательных предметов.

Следует, конечно, правильно понимать эту аналогию. Математика не является умирающей наукой и никогда ею не станет. Ее вклад в другие науки и нашу жизнь в целом велик и значителен, а логика и точность, необходимые для занятий математикой, составляют важную часть работы программного обеспечения. В самом деле, программные системы – это едва ли не олицетворение логики и точности. Программный продукт, в котором отсутствуют логика и точность, бесполезен и даже опасен.

Однако здесь возникает вопрос. Действительно ли математика лучше всего подходит для обучения программистов логике и точности? Прежде всего, насколько мне известно, никто никогда не изучал этот вопрос. Математиков, перешедших в программисты, он не интересует, а всем остальным, испытывающим благоговение перед красотой и значительностью математики, претит поднимать или изучать такой спорный вопрос. Как я уже говорил, здесь пахнет ересью.

Должен признаться: я бывший математик. Я получил диплом в одной из математических цитаделей несколько десятилетий назад. И хотя мой диплом получен давно, а знания очень устарели, я все же полагаю, что обладаю некоторой квалификацией, дающей мне право критиковать мою родную науку.

Едва начав движение от науки к практическому программированию, я заметил, сколь мало то, чему меня учили, соотносится с моей практической работой. Я проходил подготовку как «чистый» математик, среди ученых, которые, как и многие их коллеги-теоретики, свысока смотрели на любое практическое применение своей науки. А работа на производстве, куда я попал, состояла в применении математики. В той среде, где я очутился, почти все прослушанные мной курсы оказались ненужными, за исключением пары компьютерных курсов.

Все это, как я уже говорил, давняя история. Разумеется, нетрудно заметить, что математика, как ее преподавали несколько десятилетий назад, может иметь слабое отношение к современному программисту. Отсюда

следующий очевидный вопрос: может быть, для практического программирования полезнее современная математика?

Я просмотрел много книг, рекомендованных моими коллегами математиками и программистами, чтобы найти связь между их содержанием и тем, чем я занимаюсь уже больше полувека, и я нахожу эту связь весьма слабой, если она вообще имеется. Если эти математические книги, входящие в обязательный круг чтения современных компьютерных специалистов, действительно важны для их работы, то это относится к той области программирования, с которой я не имел дела. А надо сказать, что я (в отличие от своих ученых коллег) потрудился едва ли не во всех его областях!

Наверное, поэтому я и придумал «синдром латыни». Я пришел к убеждению, что мы продолжаем делать вид, будто математика связана с программированием, несмотря на то что времена, когда это действительно было так, давно миновали. Важно отметить, что я не отрицаю пользу математики для программиста. Я просто хочу сказать, что для большинства программистов эта польза очень невелика, если вообще есть. Не исключено, что изучение латыни было бы не менее полезным в смысле воспитания логики и точности.

В последние годы стало модным связывать новые идеи в программировании с математическим мышлением. Например, идея структурного программирования целиком основана на математической теории, согласно которой все языковые формы могут быть представлены посредством нескольких значимых. Другой пример – идея формальных методов, основанная на том предположении, что математические представления и доказательства являются – или должны быть – существенной частью практики программирования. Здесь имеет место некая «мощь в соучастии», что-то вроде «вины в соучастии» наоборот. Мы так часто слышали, что математически обоснованные идеи в программировании неким образом превосходят те, у которых такого основания нет, что перестали критически относиться к подобным заявлениям. Я склонен рассматривать это как «синдром непорочного зачатия». Некоторые церковники когда-то приняли ту точку зрения, что рождение ребенка девственницей в определенном отношении удовлетворительнее его альтернативного, возможно, менее безупречного варианта. Разумеется, глубокое удовлетворение вызывает вера в то, что применение компьютеров, основанное на математике, в каком-то смысле чище и лучше, чем его менее безупречная альтернатива.

Но насколько важна такая безупречность? В сложном и запутанном мире реального программирования основное правило, скорее, состоит в том, что «красиво то, что полезно». Ценность идеи в ее собственных достоинствах, а не в том, имеет она математическое основание или нет. Структурное программирование полезно, потому что позволяет создавать хорошие

программные продукты, а не потому что оно основано на математике. Если формальные методы когда-нибудь станут полезными – а на сегодняшний день они себя не проявили с хорошей стороны, – то опять-таки, потому что они дадут нам хорошие программные продукты, а не потому что основаны на чистой и красивой математике. Математика не должна превращаться в подпорку идей, ценность которых не находит подтверждения.

Стоит ли уделять этому вопросу так много места? То есть в самом ли деле все эти проблемы с математикой вредят компьютерной области или к ней надо относиться, как к латыни – раз польза есть, к чему беспокоиться о деталях? Полагаю, проблема здесь имеется.

Наши собратья-математики, развивая методологию исследований, пришли к выводу, что основными инструментами исследований являются анализ и доказательство. Если теория доказана, она правильна. Математически говоря, более сильных утверждений не существует.

Компьютерные науки применяли тот же подход к исследованиям. Я бы сказал, в основном они заключаются в том, что новая идея пристально анализируется, затем делаются некоторые попытки доказать ее корректность, после чего ее начинают пропагандировать. В этой новой области знаний доказать что-то сложнее, чем в математике. На компьютерные научные идеи влияет окружающая реальность, и доказательства неотделимы от ее несовершенства. В итоге большинство доказательств в компьютерных исследованиях оказывается просто набором доводов в пользу открытий, сделанных на стадии анализа. Получается, что в компьютерных науках слабые математические исследования выдаются за добротные.

Заявляю, что математическое исследование – просто неподходящая модель для компьютерных наук. Думаю, более подходящей можно считать ту модель, которая здесь применяется. Научный метод, как известно, состоит в формулировке гипотезы, проверке этой гипотезы и сообщении о результатах проверки. Отстаивание гипотезы должно опираться на достаточно веские данные.

В компьютерных науках редко используется научный метод. Обычно в этих исследованиях есть какие-то зачатки формулировки гипотезы, очень незначительный объем проверки и чрезмерно активная пропаганда. В результате в компьютерных науках много теорий, основанных на вере в сомнительные вещи, стремящихся игнорировать реальность и уповающих на гениальность. Не имею ничего против гениальности, но весьма опасно игнорировать реальность в сфере производства продуктов, взаимодействующих с этой реальностью и, фактически, быстро изменяющих ее.

Есть определенная ирония в том, к чему мы, кажется, пришли в поисках ответа на первоначальный вопрос. Насколько важна для компьютерных наук связь с математикой? Я бы сказал, что математические корни при-

вели компьютерные науки к такому состоянию, когда исследования здесь нельзя считать высококачественными ни с точки зрения математических принципов, ни с точки зрения научных принципов. Если учесть, что, как я доказывал в начале этого раздела, компьютерные науки больше не занимаются только «вычислениями» и при этом представляют собой и плохую математику, и плохую науку, то это не науки, а одна видимость, и их основания являются достаточно бутафорскими.

Пора уже компьютерным наукам разобраться в том, что они собой представляют. Если они хотят сохранить верность своему математическому происхождению, то должны найти новые средства, подтверждающие законность притязаний на такое наследство. Если же они хотят считаться наукой, то должны и вести себя подобающим образом. А если они будут заявлять, что занимаются «вычислениями», то должны, по крайней мере, признать, что их название сохраняется по историческим причинам, а не потому, что описывает сферу их интересов.

Все же, возвращаясь к первоначальному вопросу: насколько связь с математикой важна для компьютерных наук? Боюсь, что правильный ответ – «в очень значительной мере».

4.3. Роль интуиции в принятии решений

Современное общество склонно отвергать интуицию как метод принятия решений. Из имеющихся методов принятия решений большинство из нас в первую очередь выбрало бы количественный подход, затем рациональный, а к интуиции прибегло бы лишь в последнюю очередь.

Надо признать, что мы плохо понимаем, что такое интуиция. Мы приписываем ее женщинам и в соответствии с сексистскими традициями воспринимаем с иронией. Мы рассматриваем ее как знание, источник которого неизвестен, и не вызывающее особого доверия.

Но странное дело: когда действительно приходится принимать решения в сложных областях, все другие методы часто оказываются бесполезными. Сначала отказываются действовать количественные методы. Они прекрасны и безупречны, если задачи достаточно понятны или формулируются математически, но в нашей грубой действительности часто оказываются непригодными. Упрощая модель реального мира, чтобы иметь возможность применить эти методы, мы часто получаем приличные на вид, но не заслуживающие доверия результаты. Будем ли мы называть свои количественные методы принятия решений математическими, статистическими, научным управлением или станем исходить из числовых результатов, полученных каким-то другим способом, применение данных методов редко приносит пользу.

Рациональные методы готовы служить нам гораздо дольше. Мы собираем всю относящуюся к проблеме информацию, извлекаем из нее факторы, влияющие на решение, применяем к этим факторам критерии решения и получаем готовое решение. Звучит убедительно, четко и ясно. И действительно, если все проделать достаточно тщательно, создается полное впечатление количественного подхода. Числа, полученные в результате умозаключений, могут выглядеть в точности, как числа, полученные количественными методами! (Разумеется, здесь таится опасность, поэтому к числам нужно относиться так же критически, как к другим формам ответа.) Но даже рациональные методы могут подвести – и чаще, чем нам кажется или хотелось бы того. Когда картина осложняется политическими требованиями или крайней сложностью задачи, мы часто не ограничиваемся одними лишь рациональными методами.

Рассмотрим, например, задачу оценки сроков разработки программного обеспечения. Если полистать учебники или применяемые на практике инструкции, то можно найти очень разумные предложения относительно того, как осуществлять оценку. Эта разумность может основываться на мнениях экспертов, историческом опыте или некотором алгоритме, и во всех случаях вам предложат точное, методичное и рассудительное описание того, как нужно проводить оценку проекта.

Однако в реальном мире такой рациональный подход редко оказывается жизнеспособным. Действительные оценки – те, которые докладываются руководству, предоставляются заказчикам и отражаются в графике работ, – основываются на политических соображениях. Есть переговорный процесс, в ходе которого заказчик продукта сообщает производителю работ, к какому сроку он хочет получить результат, а производитель приводит свои разумные соображения, в результате чего (в большинстве случаев) график слегка корректируется, а затем изменяется, исходя из потребности. Не из разумных соображений, основанных на объеме работы, а из потребности – независимо от необходимого для изготовления продукта времени. Этот вывод сделан в исследовании, проведенном Ледерером [Lederer 1990], подтвержден в отчете о применении CASE-средств [HCS 1990] и многократно проверен большинством практикующих программистов.

Конечно, такие политически обусловленные графики тоже можно в известном смысле считать рациональными. Скажем, отдел маркетинга может решить, что программный продукт должен быть готов к очередной крупной компьютерной выставке, вроде бывшей COMDEX, и никакие другие соображения не влияют на дату поставки.

Но на каком основании отдел маркетинга решает, что продукт должен быть готов к следующей COMDEX? В конце концов, выставки COMDEX одно время проводились довольно регулярно – в лучшие времена дважды

в год. На чем основываются маркетологи, требуя, чтобы продукт был готов к очередной COMDEX?

Думаю, на интуиции. Исходя из своего долгого опыта работы на рынке, отдел маркетинга считает, что анонсирование продукта именно на этой очередной важной компьютерной выставке дает максимальные шансы выйти на желаемый рынок. Конечно, можно привести какие-то цифры и рациональные соображения в поддержку такой позиции, но если припереть их к стене, то они скажут, что *чувствуют* необходимость выпустить продукт к следующей COMDEX.

Такая же проблема, как с оценкой сроков, на самом деле встречается везде. Большинство людей, занимавшихся административной работой в этой отрасли, сталкивалось с проблемой следующего характера.

Начальник задает вам вопрос, ответ на который требует определенной исследовательской и аналитической работы. Вы проводите соответствующие исследования и анализ и приходите к начальнику с ответом. «Этот ответ никуда не годится, – заявляет начальник. – Идите и все переделайте». Ситуация повторяется несколько раз, пока вы не начинаете понимать, что начальник заранее знает, какой ответ ему нужен, и ваша задача в том, чтобы найти не точный ответ, а обоснования, подкрепляющие точку зрения начальника. (Конечно, сам начальник вам об этом не скажет, поскольку это выглядело бы неестественно и подозрительно.) И тогда вам приходится решать моральную проблему, с которой вы столкнулись, и либо а) дать начальнику тот ответ, который, очевидно, ему нужен, приведя доводы в его поддержку, либо б) найти выход из затруднительного положения, в котором вы оказались, – например, заставить начальника принять разумно обоснованный ответ (редкая удача!), отказаться от поставленной перед вами задачи или найти себе другое место работы.

Опять-таки, откуда ваш начальник взял свой «правильный» ответ? Из своей интуиции. Он столько времени командовал в своем бизнесе, что пришел к убеждению: он сам лучше разбирается в своей отрасли, чем все эти не от мира сего технари и аналитики, оказавшиеся в его подчинении.

Здесь возникают два важных вопроса:

1. Верен ли такой подход?
2. Что в действительности представляет собой интуиция?

Большинство из нас с подозрением отнесется к методам такого начальника. Его подход не соответствует тем образцам, на которых нас учили. Он противоречит тому, что мы знаем из учебников, и тому, что нам известно о морали. Общаясь между собой, мы высмеиваем такие задания и дивимся своему непреклонному старомодному начальнику. Как ему с такими методами вообще удалось стать начальником?

Но действительность такова, что на удивление часто начальник оказывается прав. Необычные, иррациональные способы принятия им решений в том или ином отношении приводят к успеху – ровному технологическому процессу, блестящему продукту или хорошим финансовым результатам. В чем тут дело?

Ответ на *этот* вопрос будет также ответом на второй из поставленных выше вопросов. Интуиция, как мне кажется, – это способность нашего разума использовать богатые залежи накопленной опытным путем информации (наличие которых мы можем не осознавать), применяя при этом не ясные нам методы.

Обратите внимание, что такое определение ведет нас к некоторым другим, еще более интересным выводам. Например, поскольку интуиция является функцией разума, то можно считать, что она тоже представляет собой рациональный подход. Если мы не понимаем, как она действует, то это потому, что мы не знаем, как ее объяснить, а не из-за ее иррациональности. Причина уверенности начальника в своем решении, очевидно иррациональном с виду, состоит в том, что где-то в глубине оно может оказаться рациональным.

На практике подобный способ принятия решений вызывает более позитивное отношение. Мы придумываем для него разные названия, стараясь скрыть его очевидную иррациональность за социально приемлемыми терминами. Например, мы говорим об «инстинктивном» решении, когда оно принимается на основании неких глубоких внутренних убеждений вопреки всей окружающей обстановке. По всей видимости, источником таких инстинктивных решений является наша старая знакомая – интуиция. В таком случае, «интуитивный» процесс принятия решений, обычно считающийся положительным мужским свойством, мало отличается от «женской интуиции», которую мы считаем несколько подозрительной!

В действительности, недавно (в 2005 году) вышла целая книга, посвященная принятию решений «в мгновение ока», которая весьма убедительно (с помощью приведенных результатов исследований и отдельных примеров) показывает, что моментальное решение может быть таким же эффективным, как более рациональные методы [Gladwell 2005]. Таким образом, эта идея в некоторой степени становится респектабельной!

Феномен интуитивного мышления хорошо известен практикам, но менее изучен в теории. Принятие решений на количественной и рациональной основе – это предмет, которому можно обучать и который допускает проверку; интуитивное принятие решений просто не вписывается в нашу научную систему. Поэтому я очень заинтересовался, когда несколько лет назад на ежегодной международной конференции по информационным системам докладчик из академических кругов предложил признать су-

ществование по крайней мере политических (и нерациональных) принципов принятия решений путем создания систем поддержки принятия решений, которые бы не занимались измерением, анализом и рационализацией, а принимали на входе ответ и давали на выходе обоснования в пользу такого ответа! Об осуществимости такой системы можно спорить и рассуждать часами. Однако в итоге, если учесть, что может потребоваться наделить компьютер интуицией, которая даже в человеке действует непонятным для нас образом, такая задача представляется едва ли разрешимой. Но все такой же захватывающей!

Так о чем же мы здесь говорили? И какой вывод я пытаюсь сделать? Выводов несколько, вот они:

1. Принятие решений на количественной основе, если оно возможно, обычно предпочтительнее, чем его альтернативы, но оно осуществимо не так часто, как нам того хотелось бы.
2. Принятие решений на рациональной основе – второй по значимости способ, но на практике его часто замещают чем-то другим.
3. Этим «чем-то другим» является интуиция, и даже с учетом нашего непонимания этого явления данный способ принятия решений допустим и часто применяется.

Приятно думать, что когда все другие средства бессильны, мы можем положиться на свой внутренний природный механизм принятия решений, правда?

Ссылки

GLADWELL 2005 – «Blink: The Power of Thinking Without Thinking», Little, Brown, and Co., 2005; Malcolm Gladwell.

HCS 1990 – 1990 CASE/CASM Survey, HCS, Inc., Portland, OR.

LEDERER 1990 – «Information System Cost Estimating: A Management Perspective», MIS Quarterly, June 1990; Albert L. Lederer et al.

4.4. Тьма ловушек: числа бывают разные

Есть два противоположных взгляда на роль чисел в принятии решений.

Приверженцы первого считают, что «нельзя управлять тем, чего нельзя измерить», а принимать решения лучше на количественной основе, чем на какой-либо другой.

Приверженцы второго, выступающие под лозунгом «есть ложь, грязная ложь и статистика», полагают, что количественные методы принятия решений ничем не лучше остальных, а иногда и хуже.

Кто из них прав?

Как ни смешно, скорее всего, правы и те и другие. Безусловно, лучше всего принимать решения, исходя из количественных оценок на основе честно полученных и релевантных численных данных. Но столь же безусловно, что решения, принятые на основе подтасованных или нерелевантных численных данных, не только не лучше, но и, скорее всего, хуже.

Проблема в том, что на неверные числа очень часто полагаются, не ведая того. И принятие решений на количественной основе опасно тем, что неверные цифры выглядят ничуть не хуже верных. Если я вижу число 25,623, представляющее результат каких-то измерений, то полагаю, что оно достаточно точно представляет измеряемую величину (например, среднее количество строк кода в модуле). Однако если немного поразмышлять об измеряемой величине, возникает ряд вопросов:

1. Стоит ли вообще измерять данную величину? Например, есть ли реальная польза от знания «среднего количества строк кода в модуле»?
2. Насколько точно проведено измерение? Например, сколько модулей было взято для получения этого числа? Каким образом подсчитывались строки кода? Каково было качество этих модулей? Нужно ли учитывать как хорошие, так и плохие модули?
3. Насколько значимо это измерение? Например, добавляет ли ,623 в 25,623 что-нибудь для нас сверх просто 25? Или неоправданно усиливает наше доверие к этому числу?

Взглянув на число 25,623 в таком свете и в качестве примера среднего количества строк в модуле, можно ответить так:

1. Пока мы не установили, в чем цель нашего измерения, 25,623 остается просто числом.
2. Пока мы не установили, что измерения проведены правильно, число 25,623 ничем не лучше любого другого, выбранного случайно.
3. Пока мы не установили, что все цифры корректны и релевантны, число 25,623 ничем не лучше числа 25. На самом деле, в данном примере 3/5 всех цифр не нужны («,623» только добавляет путаницы к «25»).

Иными словами, числа бывают разные. Важно уметь каким-то образом различать хорошие и плохие числа.

Дело еще осложняется тем, что сторонники количественного подхода открыли лозунговую кампанию с целью убедить в своей правоте доверчивых руководителей программных проектов:

> «Измерить – значит понять». Джеймс Кларк Максвелл
>
> «Нельзя управлять тем, что нельзя измерить». Том Демарко
>
> «В невидимую цель трудно попасть». Том Гилб

Но война лозунгов – вещь обоюдоострая. Вот другие лозунги, взятые из [Leveson 1993]:

> «Данные оценки рисков можно сравнить со сведениями, полученными от пленного: если достаточно долго мучить его на допросе, он скажет все, что вы хотели бы услышать». Уильям Раккельсхауз, руководитель Национального агентства США по защите окружающей среды (два срока)

> Игры с числами при оценке рисков «можно проводить только закрытым образом между взрослыми лицами, достигшими взаимной договоренности, иначе они могут быть неправильно интерпретированы». И. А. Райдер, Комитет по здравоохранению и промышленной безопасности Великобритании

> «Наш энтузиазм в отношении измерений не должен распространяться на то, что измерений не допускает». Нэнси Ливсон и Кларк Тернер

Как это обычно бывает, приверженность к лозунгам и фанатичная пропаганда оказываются плохой базой для любых методов принятия решений, будь они количественные или нет.

Приведенные высказывания не следует рассматривать как неодобрение принятия решений на количественной основе. Сейчас появилось достаточно отличных книг по метрикам в программировании, написанных такими авторами, как Роберт Грэйди (Robert Grady), Билл Хетцел (Bill Hetzel) и Норм Фентон (Norm Fenton), и организациям, занимающимся программированием, стоит всерьез задаться вопросом: «Не следует ли нам внедрить подход на основе метрик, чтобы усилить количественный аспект нашей процедуры принятия решений?»

Но будьте осторожны. На этом пути вас поджидает множество опасностей.

Ссылки

LEVESON 1993 – «An Investigation of the Therac-25 Accidents», *IEEE Computer*, July 1993; Nancy G. Leveson and Clark S. Turner

Глава 5 | Процесс или продукт?

Введение
5.1. Приводит ли хороший процесс к лучшему продукту?
5.2. Приводит ли хороший процесс к лучшему продукту? Другая точка зрения
5.3. В шаге от славы
5.4. Разные мысли по поводу процесса программирования
5.5. Процессы и люди: путь к хорошему продукту
5.6. Оценка результатов применения модели зрелости процесса (CMM)
5.7. Когда лучше ориентироваться на продукт, а когда – на процесс?

Введение

Новшества, вносимые в технологический процесс, лишь упрощают существующую систему представлений,... не требуя внесения в нее радикальных изменений. Но революция в науке резко оспаривает установившуюся систему представлений и фактически заменяет ее новой.

Ной Кеннеди

«The Industrialization of Intelligence»,
Unwin Hyman, 1989

Вот мы и добрались до популярной темы!

Когда последние годы предыдущего тысячелетия сходили с наших календарей, в которых впервые появились цифры «200x», какая проблема программирования вызывала самые горячие дискуссии?

Технологический процесс создания программ. Благодаря «модели зрелости процесса создания ПО» (Capability Maturity Model, CMM), разработанной Институтом программирования SEI, мы стали исследовать, проверять и оценивать процессы так, как не делали этого никогда ранее.

Компании стали выдвигать странные лозунги вроде «зрелость 3 к '03» или «4 к '04». Понимать это нужно было как «мы стремимся достичь уровня зрелости x к 200x году». И они очень старались, эти компании. Поговаривали, что от уровня зрелости компании может зависеть количество контрактов, которое она получит. Выживание в условиях конкуренции и организация технологических процессов вдруг слились воедино в умах множества людей из множества программистских компаний.

Но погодите, ребята! Боюсь, мы слишком увлеклись этой идеей. Прошло уже больше половины первого десятилетия нового века, мы глубоко уверовали в то, что технология – мерило успеха в программировании, а между тем остается нерешенным фундаментальный вопрос: действительно ли хорошая организация технологического процесса приводит к появлению хороших программных продуктов?

Этот вопрос должен вызвать у нас сразу множество мыслей. Во-первых, какова наша цель – процесс или продукт? (Практики программирования ответят быстро и уверенно: конечно, продукт. Некоторые теоретики не столь категоричны.)

Допустим, действительно, цель – продукт. Тогда сразу другой вопрос: какие еще пути, помимо процесса, ведут к хорошему продукту? Или: откуда известно, что именно этот предложенный SEI процесс дает лучшие результаты? И, конечно, есть еще важный вопрос, вытекающий из ключевого вопроса предыдущего абзаца: откуда нам известно, что хороший процесс приводит к хорошему продукту?

Поставив все эти вопросы, сделаем кое-что, прежде чем начать с ними разбираться. Во всех введениях предыдущих глав мы определяли по два ключевых термина, вокруг которых строился весь материал. Так же мы поступим и здесь.

Процесс – это «последовательность действий или операций для создания, изготовления или достижения чего-либо; ряд последовательных изменений, протекающих естественным образом; ход событий».

Продукт – это «нечто, создаваемое природными процессами, или в сельском хозяйстве, или в промышленности, или как следствие».

(Попутно отмечу, что промышленность довольно часто упоминается в определениях XXI века. Похоже, словари не спешат фиксировать переход от Америки фабричных труб к Америке обработки информации.)

Как бы то ни было, что нам делать с этими вопросами?

Верно ли, что хороший процесс приводит к хорошему продукту? По правде говоря, мы этого не знаем. Интуиция подсказывает, что так должно быть. Однако работ, в которых делалась попытка ответить на этот вопрос, довольно мало. Попытка SEI дала неопределенные результаты (об этом будет рассказано в разделе 5.3). Следующие разделы так или иначе отвечают на данный вопрос.

Какова наша цель – процесс или продукт? Типичный ответ практика мы уже видели. Разумеется, продукт; все остальное неважно. Полагаю, большинство согласится с этим. У меня есть друг, который увлекается генеалогией. Он очень любит часами сидеть в библиотеке, сосредоточенно изучая пыльные старые регистрационные книги и пытаясь составить фрагменты семейной истории, как пазл. Но как бы он ни любил разыскивать записи в этих книгах, главная его цель – результат. Он хочет проследить корни этой семьи глубже, чем смог кто-либо до него. И когда он попадает на тупиковую ветвь, где не может отыскать предка для вершины изучаемого им фамильного древа, все это написано на его лице. В такой момент он готов отдать все лучшие процессы на свете за один хороший продукт.

Какие еще пути, помимо процесса, ведут к хорошему продукту? Таков был еще один вопрос, поставленный выше. И здесь есть много ответов.

Хорошие работники – это путь к хорошему продукту. (Многие скажут даже, что хорошие работники важнее хорошего процесса.) Обложка книги

Барри Боэма «Software Engineering Economics» (Экономика программного инжиниринга) явно стремится продемонстрировать, что хорошие люди в значительно большей мере, чем процесс, определяют эффективность программных разработок. Билл Кертис, сменивший Уоттса Хамфри в качестве руководителя исследований SEI по процессам, добавил в модель зрелости ориентацию на работников.

Хорошая технология – тоже путь к хорошему продукту. (Модель зрелости описывает процессы управления, а не технологические процессы.)

Умелое заключение контрактов и ведение переговоров – тоже путь к хорошему продукту.

У продукта, как и у стола, несколько ножек, и ясно, что процесс – лишь одна из них.

Откуда известно, что именно предложенный SEI процесс дает лучшие результаты? Это следующий из поставленных нами выше вопросов. Честный ответ: нам это неизвестно. Не было серьезных исследований, сравнивающих модель SEI с другими. (Хотя Каперс Джонс в своей последней книге утверждает, что он проводил такие исследования, и они показали, что его модель лучше, чем модель SEI!) Но мы можем уверенно сказать, что метод SEI, пожалуй, лучше отлажен, исследован и развит, чем прочие из имеющихся моделей. Для изучения модели приглашались исследователи со всего света. Их предложения были учтены. Здесь, пожалуй, уместно применить одну из главных идей этой книги: едва ли когда-либо будет найдено оптимальное определение процесса разработки программного продукта, а потому (как показано в одной из предыдущих глав) следует удовлетвориться разумно достаточным. И вспомним один из главных критериев разумной достаточности... Является ли модель SEI приемлемой? Ответ, конечно, утвердительный.

И последний из наших вопросов, отчасти повторяющий первый: откуда нам известно, что хороший процесс приводит к хорошему продукту? Это, конечно, ключ ко всей этой главе, и мы неоднократно обсудим его ниже.

Итак, читайте дальше. Вы встретите несколько противоречащих друг другу ответов на наш главный вопрос, но не исключено, что из этого хаоса возникнут и какие-нибудь разумные мысли. Например, в последнем из разделов излагается довольно простой взгляд на то, когда вместо того чтобы ориентироваться на процесс, следует (увы, лишь с моей точки зрения) ориентироваться на продукт.

Вот вам интересная тема для размышлений. В чем смысл чтения этой книги – получить удовольствие, путешествуя от первой страницы к последней? (Это процесс.) Или в том, чтобы узнать что-то новое? (Это продукт.) В любом случае, желаю вам успеха!

5.1. Приводит ли хороший процесс к лучшему продукту?

Мы живем в эпоху, когда процесс (метод создания продукта) считается более важным, чем сам продукт.

Специалисты по компьютерным наукам занялись совершенствованием процесса, основываясь на том предположении – иногда об этом говорится прямо, – что, улучшая процесс, мы, несомненно, улучшим продукт. На этом предположении, например, основана хорошо известная модель зрелости процесса SEI. Множество компаний по всему миру тратит массу средств на изучение и совершенствование своего процесса создания ПО. Они могут делать это, рассчитывая, что более высокая оценка по 5-балльной шкале SEI принесет им больше контрактов и прибыли, или искренне полагая, что лучший процесс дает лучший продукт, но, в любом случае, они это делают.

Но верно ли, что процесс может улучшить продукт? Есть ли какие-нибудь исследования по этому предмету? Как было бы хорошо написать здесь, что есть ряд исчерпывающих исследований, результаты которых опубликованы и не оставляют никаких сомнений в истинном положении вещей! К сожалению, какие-либо подобные исследования мне неизвестны. Изучались некоторые второстепенные аспекты нашего главного вопроса, но я не знаю ни одного исследования (и непонятно, как его можно было бы провести), которое раз и навсегда прояснило бы суть вопроса. И SEI, понимая, что необходима какая-то проверка CMM, до сих пор не смог организовать ничего подобного.

С учетом этого, обратимся к исследованиям, касающимся побочных аспектов нашего вопроса. Мне больше всего нравится статья Сасы Деклевы (Sasa Dekleva [Dekleva 1992]), основанная на обзоре мнений профессионалов в области информационных систем, – вероятно потому, что она содержит факты, которые одновременно согласуются с интуицией и противоречат ей. Вопрос, который Деклева хотел выяснить в своем исследовании: влияет ли на легкость сопровождения продукта применение его разработчиками передовых, усовершенствованных методов программирования? Это достаточно близко к сути нашего вопроса: приводит ли усовершенствование процесса разработки к продукту, который легче сопровождать?

Интуиция – во всяком случае, моя, но полагаю, что и ваша, – просто кричит о том, что ответ очевиден. Несмотря на всю прежнюю болтовню о сравнительной важности процесса и продукта, мы в общем-то не сомневаемся, что усовершенствование процесса разработки положительно влияет на качество конечного продукта. Уверенность в этом настолько

сильна, что может даже возникнуть недоумение – стоило ли вообще проводить такое исследование?

Но то, что Деклева выяснил в результате, изучая работу руководителей программных проектов, поразило его самого. Опишем, как было организовано его исследование.

Прежде всего, Деклева изучил опубликованные отчеты исследований, посвященных влиянию усовершенствований процесса разработки на производительность труда. То, что он выяснил, стало для него первой неожиданностью. Оценочных исследований вообще оказалось мало (Весси и Вебер [Vessey 1984] столкнулись с тем же самым несколькими годами ранее, когда искали количественные оценки эффективности структурного программирования). При таком множестве людей, предлагающих и пропагандирующих новые технологии разработки программного обеспечения, практически никто не отваживается оценить их количественно. (Такое же заключение, но в более общем виде, было сделано в более поздней работе [Fenton 1993].) А то, что Деклеве все же *удалось* найти, свидетельствовало об отсутствии роста производительности. (Один из авторов [Banker 1991] написал: «Многие из предполагаемых преимуществ применения детализированной методологии, требующей создания обширной документации, никак не проявляются до последующих проектов, в которых требуется внести в систему исправления или усовершенствования». Деклева с усмешкой прокомментировал это: «Иными словами, если современные методы разработки ИС не повышают производительность при создании продукта, остается надеяться, что они проявят свои положительные качества во время сопровождения».)

Движимый такими результатами, Деклева опросил 112 руководителей информационных систем компаний из списка Fortune 500, желая выяснить, как же действительно *отразился* процесс разработки на ходе сопровождения продукта. Он достаточно конкретно определил, какие процессы разработки считать усовершенствованными, указав, что они должны: а) основываться на принципах разработки ПО, б) использовать принципы проектирования информационных систем, в) включать в себя создание прототипов и г) применять CASE-средства. Он разослал свой опросник и стал ждать результатов.

Когда поступили результаты, в них, в целом, не оказалось ничего неожиданного. В большинстве случаев то, что руководители сообщили Деклеве, вполне соответствует нашим интуитивным ожиданиям. Программы, разработанные с применением усовершенствованных технологий, были надежнее и требовали меньше исправлений.

Однако, как я отмечал выше, сюрприз все же был. Подсчитав общий расход времени на сопровождение для усовершенствованных методов разра-

ботки, Деклева обнаружил, что улучшенные методы повлекли *рост* суммарной стоимости сопровождения. При этом схема была ясна. Стоимость, как и предполагалось, была ниже в течение первого года работы системы или около того, но после резко возрастала. Чем объяснялась эта неожиданная тенденция? И действительно ли из нее следовало, как казалось на первый взгляд, что усовершенствованный процесс ведет к созданию продукта, который труднее сопровождать, и, следовательно, имеющего более низкое качество?

Деклева поработал с полученными данными еще, и проявилась более понятная картина. Рост стоимости сопровождения был обусловлен не исправлением ошибок, а внесением в продукт усовершенствований. По-видимому, пользователи правильно разрабатывавшейся системы, несмотря на то что она требовала меньше исправлений, спешили воспользоваться возросшими возможностями модификации системы и требовали внести в нее изменения. А это означало, что согласно полученным данным благодаря усилиям, направленным на совершенствование процесса, продукт становился не хуже, а лучше, даже с учетом этого неожиданного фактора роста общей стоимости сопровождения.

Сделаем паузу и вернемся к изначальному вопросу. Что говорят нам эти данные о значении процесса и продукта? Действительно ли улучшение процесса ведет к улучшению продукта?

Ответом, если немного подумать, будет решительное ДА. Что касается количества ошибок в продукте, здесь все просто и понятно. В отношении усовершенствований продукта положительный ответ получен косвенным и не очевидным образом. И, тем не менее, ответ – ДА. Если система настолько проста в сопровождении, что заказчики и пользователи часто просят внести в нее изменения, то она явно лучше такого продукта, который никто не хочет изменять, потому что изменения обходятся слишком дорого.

В таком случае, это исследование, цель которого, по моему мнению, не вполне совпадает с сутью нашего вопроса о связи процесса с продуктом, все же подсказывает нам положительный ответ на рассматриваемый вопрос. Результаты могут оказаться неожиданными, как это случилось с самим исследователем, но, тем не менее, они несколько подкрепляют идее о том, что улучшение процесса приводит к улучшению продукта.

В следующих разделах мы еще глубже изучим этот вопрос.

Ссылки

Banker 1991 – «A Model to Evaluate Variables Impacting the Productivity of Software Maintenance Projects», *Management Science,* January 1991; R.D. Banker, S.M. Dater, and C.F. Kemerer.

DEKLEVA 1992 – «The Influence of the Information Systems Development Approach on Maintenance», *MIS Quarterly,* September 1992; Sasa M. Dekleva.

FENTON 1993 – «How Effective Are Software Engineering Methods?» *Journal of Systems and Software,* August 1993; Norman Fenton.

VESSEY 1984 – «Research on Structured Programming: An Empiricist's Evaluation», *IEEE Transactions of Software Engineering,* July 1984; I. Vessey and R. Weber.

5.2. Приводит ли хороший процесс к лучшему продукту? Другая точка зрения

В предыдущем разделе (озаглавленном почти так же, как этот) мы исследовали вынесенный в заголовок вопрос и получили положительный, хотя и неожиданный ответ.

Но взглянем на этот вопрос с другой стороны. Как уже было сказано, хотелось бы представить ряд экспериментальных результатов, демонстрирующих правильность той точки зрения, согласно которой улучшение процесса ведет к улучшению продукта, но свидетельств в ее пользу маловато. Обращаю внимание, что, задавая такой вопрос, мы не намекаем на то, что хороший процесс *не* ведет к хорошему продукту, а лишь считаем, что эта тема достойна обсуждения.

Ввиду недостаточности фактических свидетельств можно при изучении этого вопроса обратиться к мнению экспертов, и тут мы встретим резкие высказывания с обеих сторон.

Поскольку предыдущий раздел отражал достаточно позитивную точку зрения по этому вопросу, представим теперь противоположные взгляды. В весьма гневной статье, много лет назад опубликованной в IEEE Spectrum [Frosch 1969], Роберт Фрош, бывший в ту пору помощником министра ВМС США и руководящим работником одной из крупнейших исследовательских организаций США, обрушился на положительную точку зрения с резкой критикой.

Он начал статью с краткого изложения своих взглядов: «Я полагаю, главная трудность в том, что нас настолько заворожила... технология, что мы мыслим исключительно категориями процедур, систем, контрольных рубежей, графиков, диаграмм ПЕРТ, систем обеспечения надежности, управления конфигурациями, групп сопровождения и прочих мелких канцелярских средств «системного инженера» и администратора... В результате мы получаем разработки, в которых соблюдены все правила, но нет никакого результата».

Крепко сказано. Фрош выстроил все методологии процесса, какими только можно воспользоваться, независимо от области и расправился с ними одним ударом! Но есть ли у него сколь-нибудь глубокие рассуждения в поддержку своего вступительного выпада?

Есть, целых пять страниц. Предсказывая неудачу таких концепций, как каскадный (однонаправленный) жизненный цикл, далее он пишет: «Отличными примерами служат диаграмма ПЕРТ и график с контрольными точками. В обоих случаях фактически предполагается, что разработка и проектирование происходят путем последовательного выполнения сначала шага *A*, затем шага *B*, затем шага *C* и так далее. Всякому, кто когда-либо занимался разработкой или проектированием (а не организовывал систему управления ими), хорошо известно, что в реальном мире развитие происходит в виде итеративного процесса с обратной связью, который больше похож на спираль, чем на прямую линию». (Намек на спиральный жизненный цикл, и это в 1969 году!)

Фрош продолжает: «Процедуры просто игнорируют... характер окружающего нас мира, и технологический процесс низведен до уровня канцелярских отчетов. В значительной степени это приводит к тому, что руководители проектов, вместо того чтобы действовать согласно реальным требованиям решаемых ими задач, стараются удовлетворить требования применяемых средств управления. Очевидно, что это абсурд».

Итак, Фрош сокрушает прежние методы управления, обреченные на провал, как мы выяснили двадцатью годами позже него. Но справедлива ли его критика и в отношении сегодняшних технологий управления?

Думаю, что да. Вот следующая реплика Фроша: «Мы дошли до того, что достичь отдельных показателей стало важнее, чем создать работающую систему... Вместо реального наблюдения за ходом разработки занимаются отслеживанием прохождения примитивного набора заранее определенных контрольных точек...

Единственный действенный метод, который я знаю, – это найти толкового человека с помощниками, которые смогут разобраться в задаче – не в спецификации ее, не в каком-то уже написанном сценарии, а в том, что творится в головах тех, перед кем стоит задача, которую нужно решить...».

Хотя и в этом отрывке Фрош целится конкретно в управление на основе контрольных точек, можно понять его шире – что чрезмерное внимание, уделяемое процессу, ослабляет наши возможности создания хорошего продукта.

Правильна ли такая оценка точки зрения Фроша? Думаю, что да. Вот что он говорит в конце своей статьи: «Мы забыли, что инженерное дело является искусством... Нужно вернуть в инженерное дело ощущение творчества

и вдохновения. Талант, знания и энтузиазм – вот качества людей, которым можно доверить инструменты; тем, кто лишен этих качеств, обычно не помогают ни технологии, ни инструменты».

Вот какое существует иное мнение по поводу значения процесса для создания хорошего продукта. Есть такой автор, который считает, что очень часто хороший процесс мешает созданию хорошего продукта. Убедительность его слов сильнее моей интуитивной реакции. Полагаю, Фрош говорит о необходимости проявлять крайнюю осторожность в обращении с процессом. И никогда нельзя терять ориентацию на продукт, а также недоучитывать роль тех, кто его создает.

В этой статье высказываются также определенные мнения по другим, близким вопросам:

- Фрош поддерживает ту точку зрения, что «спецификации требований вредны» (поскольку они стремятся зафиксировать то, чему внутренне присуща гибкость, – он говорит, что «идея полной спецификации абсурдна»).
- Он поддерживает точку зрения, что правильный подход к решению задач – «сначала сложная часть», а не «сверху вниз».
- Он поддерживает «разумную достаточность» (satisficing – поиск работающего и приемлемого решения) в противовес «оптимизации» (поиску лучшего из возможных решений) – «оптимизация может просто стать описанием того, какую катастрофу вы хотите на себя навлечь».

Ссылки

Frosch 1969 – «A New Look at Systems Engineering», *IEEE Spectrum,* September 1969; Robert A. Frosch.

5.3. В шаге от славы

Некоторое время назад я рецензировал одну исследовательскую работу, оказавшуюся весьма важной, несмотря на односторонний взгляд ее автора. Автор, увлекшись одним направлением исследований, упустил из виду глубокие возможности, открывавшиеся в другом направлении. В результате наша дисциплина много потеряла.

Я расскажу вам об этой статье – чего добивался ее автор, и что могло бы получиться в ином случае.

Автор хотел написать статью по результатам исследования преимуществ, которые приносят структурный анализ и проектирование на стадии сопровождения. То есть ставился вопрос: если программный продукт изготовлен с соблюдением правил структурного подхода, будет ли его легче

сопровождать, и если да, то насколько? Конечно, это важный вопрос, имеющий первостепенное значение в нашей области. Его нужно было задать еще лет тридцать назад, когда структурные методы только начинали ускоренными темпами внедрять в учебные расписания и практику. Но лучше исправить ошибку поздно, чем никогда. Намерения автора были похвальными и важными.

Метод исследования был любопытен. Авторы, узнав о том, что государство изменило некоторые законы, задействованные в программном обеспечении большинства предприятий (не помню в точности, о чем шла речь – возможно, о существенных изменениях в порядке взимания налогов с заработной платы), обратились в ряд предприятий с такими вопросами:

1. Сколько времени/средств потребовалось для внесения изменений?
2. Применялись ли при разработке модифицируемых программ структурные методы?

Если бы все прошло гладко, автор смог бы выявить простую и устойчивую связь между применением структурных методов и легкостью выполнения данной модификации, проследив ее на примере различных программ на различных предприятиях, и оценить ее количественно. Вывод вроде «программы, созданные структурными методами, на 5 (или 35, или 150, или сколько-то там) процентов легче модифицировать, чем созданные без их применения» стал бы важным и замечательным вкладом в нашу науку.

Однако на деле вышло иначе. Прежде всего (запомните этот момент, мы к нему еще вернемся), время, затраченное на модификацию, очень сильно варьировало. Одни системы исправлялись за несколько часов, другие потребовали работы, измеряемой человеко-месяцами. Что было еще важнее для данного исследования, так это отсутствие корреляции между длительностью модификации и применением структурных методов во время разработки.

Таким образом, автор сделал в своей статье следующий вывод: применение структурных методов не приводит к сколь-нибудь ощутимой разнице в усилиях, необходимых для осуществления (данной конкретной) модификации программных систем.

Конечно, это важное открытие, даже если многие из нас будут им несколько раздосадованы. Думаю, многие из нас где-то в глубине души верили, что структурные методы все же облегчают создание и модификацию программ. Чего большинство из нас *не знает*, если честно, так это *в какой мере* они это делают. И автор статьи сделал открытие, которое, с одной стороны, бросается в глаза, потому что противоречит интуиции, а с другой стороны легко забывается, потому что мы внутренне с ним не согласны.

Но вспомните начало этого раздела, где я сказал, что автор, по моему мнению, упустил крупное и даже более важное направление, куда могло повернуть его исследование. Выше я намекнул читателю, что мы еще вернемся к обсуждавшемуся моменту. Совпал ли ход наших мыслей?

Вот в чем, по моему мнению, состоит неудача автора. Помните, как на одних предприятиях достаточно было нескольких часов, чтобы внести требуемые правительством изменения, а другим на это понадобились человеко-месяцы? Задумайтесь над этим. Есть программные системы, которые вроде бы выполняют идентичные функции, но время, необходимое для внесения в них одинаковых изменений, резко различается. И ошибка автора в том, что он не выяснил причины этого.

Он ограничился выводом: «Структурные методы не являются причиной». Но что-то же *является*! Что же именно?

Тут я хочу сделать несколько замечаний относительно методов и целей исследования, потому что, как мне кажется, это очень важно для понимания происходящего.

В этом исследовании автор применил то, что некоторые ученые называют «подходом, основанным на теории». Он напоминает научный метод, предполагающий формулирование одной или нескольких гипотез исходя из некоторой теории и последующую проверку этих гипотез и, таким образом, теории, на которой они основаны. Это совершенно замечательная форма исследования, потому что она обращается к самой сути нашего интереса (проверка базовой теории) и делает это упорядоченным образом (путем оценочного изучения). Любопытно, что этот метод гораздо чаще применяется в области информационных систем, чем в компьютерных науках.

Задачу исследования я бы охарактеризовал как «оценку процесса». Иными словами, автор предположил, что структурные методы, применяемые в процессе создания программного обеспечения, должны оказать влияние на длительность/стоимость сопровождения, и решил проверить гипотезу о том, что это влияние должно быть положительным. В добрый путь.

Продукт, а не процесс

Я считаю, что автор остановился в своем исследовании на полпути (и, вероятно, не понял этого), получив данные о том, что рассматриваемые системы глубоко различались по своему характеру и выполняемым функциям. Чем еще можно объяснить такой значительный разброс затрат времени на модификацию? Из этого следует важный вывод: *различия, обнаруженные в результате исследования, обусловлены не процессами, в которых создавались продукты, а самими этими продуктами*.

Сформулирую свою мысль еще раз, но иначе. Автор решил измерить эффект от процесса. В результате исследования существенных эффектов от процесса обнаружено не было, но сами продукты оказались глубоко различными. И я считаю, что, завершив изучение процессов, автор должен был перейти к следующему вопросу: какие свойства этих программных *продуктов* проявились в обнаружившихся больших различиях данных?

Исследовать, не ограничиваясь проверкой теории

Не знаю – может быть, автор заметил, что произошло, и сразу начал другое исследование, направленное на новую проблему. Скорее всего, нет. Я исхожу здесь из того, что принципы исследования были такими, как я описал выше. Напомню, что это исследование основывалось на теории. Но попутно, как мне кажется, автор обнаружил то, что не укладывалось в рамки этой теории. Речь идет о различиях в продуктах, и для их изучения потребовался бы совершенно иной метод, эмпирическое исследование имеющихся программных продуктов, при проведении которого может отсутствовать представление о будущих результатах. Конечно, *есть* методы проведения и таких исследований, и именно они нужны были в данном случае. Но некоторые смеются над такими методами, называя их «походом на рыбалку», – и все потому, что исследователь не выдвинул предварительную гипотезу относительно того, что может быть обнаружено, исходя из имеющейся у него теории.

Данный раздел получился не вполне последовательным, поскольку начался с рассказа об исследовании, в котором выяснились неожиданные факты, но не удалось дать им глубокое объяснение. Поэтому в конце я хочу немного навести порядок, подытожив несколько моментов, которые *я* считаю здесь важными:

1. Стало модным изучать процесс и его влияние на конечный продукт. Однако есть основания полагать – и данное исследование прекрасно это показало, – что связь между процессом и продуктом в лучшем случае не отчетлива, а в худшем – попросту отсутствует. Изучать продукты гораздо труднее, чем процессы, но я считаю, что мы недалеко уйдем в изучении процесса, если сначала хорошо не разберемся в том, какую роль играет сам продукт.
2. Стало модным проводить исследования на базе имеющейся теории, и это, конечно, важный метод исследований. Но в иные времена – и это относится к такой новой дисциплине, как системы и программное обеспечение, – разрыв между теорией и практикой настолько велик, что теория, объясняющая факты, может еще просто не существовать. В таких случаях исследования, основанные на теории, оказываются бесполезными, и прогресс может быть достигнут только с помощью

поисковых исследований, задачей которых и является построение теорий, подвергаемых проверке позднее.

3. Нам всем следует сделать важный вывод из неудачи этого исследования. Я, например, понятия не имею, куда нас мог привести предлагаемый поворот в этом исследовании, но мой совет всем исследователям: если, действуя в избранном направлении, вы обнаружили факты, которые не можете объяснить, остановитесь и внимательно посмотрите по сторонам – не исключено, что вы наткнулись на нечто более важное, чем то, что собирались изучать первоначально.

5.4. Разные мысли по поводу процесса программирования

Часто интересные идеи по поводу программирования обнаруживаются весьма неожиданным образом.

Конечно, рассчитываешь найти такие идеи в научных статьях из специализированных журналов. Но мне, например, доставляет особое удовольствие найти какие-то идеи по поводу программирования в не столь очевидных источниках.

Вот, например, цитата: «Целью должен быть результат, а не процесс». Не правда ли, она звучит так, будто взята из некоего журнала по практическому программированию, решившего поспорить с моделью зрелости процессов SEI?

Отнюдь! Эта цитата не откуда-нибудь, а из «The Wall Street Journal». Статья называется «Why Smart People Do Dumb Things» (Почему умные люди делают глупости), 21 декабря 1992 года, автор Мортимер Файнберг (Mortimer R. Feinberg), и размещена она в колонке «Manager's Journal» (Дневник руководителя).

Суть статьи: сверходаренными людьми нужно управлять иначе, чем обычными. Но как именно? Статья дает несколько ответов на этот вопрос.

Суть приведенной выше цитаты: сверходаренные люди не всегда поступают так же, как все остальные. Не пытайтесь насильно обуздать их творческий гений с помощью смирительной рубашки формализованного процесса. Но когда они в итоге что-то сделают, обязательно убедитесь, что их *продукт* имеет ценность. Также следите за *их* продвижением к результату с помощью средств контроля процесса, оправданных в конкретных обстоятельствах.

Любопытно, особенно с учетом источника.

А вот еще один пример интересной (и, как оказалось, аналогичной) идеи, почерпнутой из неожиданного источника. Ниже приведены отрывки из письма – в чем и заключается своеобразие этого источника – в редакцию одного из наших технических журналов, «Communications of the ACM». Письмо написал Эдвард Руэт (Edward S. Ruete), оно опубликовано в августовском номере «CACM» за 1990 год и было реакцией на очень интересную статью Джонатана Грудина (Jonathan Grudin) с интригующим названием «The Case Against User Interface Consistency» (Иск против единообразия интерфейсов). В этой статье автор утверждал, что проектировать интерфейс следует руководствуясь не только единообразием, но и потребностями приложения, и что учет этих потребностей может *нарушать* условие единообразия интерфейсов.

В своем письме Руэт в основном соглашался с Грудином. Попутно он изложил ряд мыслей, которые хорошо согласуются с приведенной выше цитатой из «The Wall Street Journal». Руэт провел аналогию между разработкой программного обеспечения и творчеством писателя, сказав, что «неважно, как вы делаете это, если получается хорошо; если принять эту точку зрения, то инструментарий и технологии для разработки программ предстают в ином свете... они хороши лишь постольку, поскольку их применение приводит к выпуску хороших программ».

А в заключительном абзаце Руэт замечает: «Вопреки всем стараниям теоретиков программирования, включая лично меня, убедить нас в противном, программирование является по своей сути творческой деятельностью. Стандарты могут гарантировать нам выпуск хорошей программы с таким же успехом, с каким они гарантируют появление хорошей пьесы, романа или статьи об искусстве общения».

Польза от инструментов и стандартов в программировании – ограничена. Любопытная свежая мысль.

И третий пример материала из неожиданного источника. На этот раз из статьи, представленной в журнал «Journal of Systems and Software», в котором она была опубликована после странных превращений. Первоначальный вариант назывался «Controllable Factors for Programmer Productivity...» (Управляемые факторы производительности труда программистов...) и принадлежал Али Мили (Ali Mili) и еще нескольким авторам. «В отличие от всех других видов промышленного производства, производительность труда в программной индустрии... в очень малой степени может быть повышена в результате применения инструментов программирования. Согласно измерениям, которые провел Боэм [в «Software Engineering Economics», 1981], разница в продуктивности между программистом, который вообще не пользуется никакими инструментами, и тем, который применяет самые современные и мощные инструменты... не пре-

вышает 50%. Чтобы правильно оценить эту цифру, представьте себе разницу в производительности труда при сборке автомобилей, если в одном случае применяются только отвертки, а в другом – самые современные роботы».

И снова мы встречаемся с мнением, что инструменты (а значит, процесс) приносят в программировании ограниченную пользу. Однако контекст этого утверждения здесь особенно интересен.

Наконец, четвертый материал. На этот раз, реплика в IEEE Computer в связи с дискуссией по поводу значения стандартов для процесса создания программного обеспечения.

Автор реплики (IEEE Computer, май 1993 года) Норман Шнейдевинд (Norman F. Schneidewind), но в основном он цитирует участников. Вот одна особенно интересная цитата в связи с рассматриваемым здесь вопросом:

> Билл Дитрих (Bill Dietrich) из Атланты написал: «Широко поддерживается принцип производства, согласно которому результатом хорошего процесса является хороший продукт. А насколько общепризнанно то, что результатом хорошего процесса проектирования является хороший проект? Настолько ли хорошо изучен процесс проектирования, что он допускает стандартизацию?
>
> ...Меня беспокоит, что сторонники стандартизации процесса могут неправильно представлять себе фундаментальную сущность разработки программного обеспечения. Понятно их желание сделать процесс разработки программ измеримым, воспроизводимым и надежным. Но разработка программ по большей части состоит из проектирования... Какие есть свидетельства в пользу того, что проектирование является измеримым или воспроизводимым видом деятельности? Каковы критерии оценки качества проекта?»

Вот вам четыре интересные точки зрения из четырех совершенно разных источников. Но особенно интересно то, что они, как мне кажется, говорят об одном и том же. Перефразируя, предлагаю такую формулировку. Процесс – это не универсальное средство для решения задач программирования. Еще более важно, что это не панацея. Тот, кто считает иначе, просто не желает замечать имеющиеся факты.

5.5. Процессы и люди: путь к хорошему продукту

Хорошие люди – путь к хорошему продукту. На самом деле, многие чувствуют, что для создания продукта люди гораздо важнее, чем процесс.

Что это – малосущественное философское расхождение или здесь не исключены практические последствия?

Было уже два случая, когда IBM, намереваясь создать новую операционную систему (сначала OS/360, затем OS/2), нанимала для этой работы сотни людей. Насколько успешным оказался этот метод в первом случае, прекрасно описано в классической книге «Мифический человеко-месяц» Фреда Брукса[1]. Не слишком успешным. Что касается OS/2, то конечный результат оказался еще хуже.

(Интересно отметить, что модель зрелости процесса SEI – вершина внимания к процессу – уходит корнями в IBM, где некоторые из ее основных архитекторов трудились еще над OS/360.)

Когда к работе над программным проектом привлекается столько людей, большое значение приобретает хорошая организация процесса. Управление сотнями программистов для достижения общей цели становится важной самостоятельной задачей. Для этого и существует процесс. Ничего удивительного в том, что IBM вкупе с Уоттсом Хамфри и его последователями уделяют такое внимание процессу.

Но нет ли иных путей? Если судить по другим программистским фирмам, поменьше, то есть. Эти компании не могут позволить себе создавать программы методом монголо-татарской орды. Они *вынуждены* искать другие пути. И этот другой путь они находят в выборе сотрудников.

Если IBM привлекает сотни людей и организует процесс, в рамках которого те разрабатывают операционную систему, то в малых программистских компаниях от силы несколько десятков сотрудников. Как же им удается справляться с работой? Путем тщательного подбора этих нескольких десятков. В некоторых компаниях тщательно проверяют кандидатов на работу (вплоть до представления ими вместе с другими документами кода написанных ими программ, который они считают своим достижением!) и освобождаются от недостаточно эффективных работников (по результатам оценки их вклада коллегами). Отбор хороших работников – от найма до увольнения – играет решающую роль для жизнеспособности маленьких программистских компаний.

Если исходить из предположения, что правильный подбор сотрудников – гораздо более экономичный способ получения хорошего продукта, чем правильный процесс, остается лишь выяснить, не страдает ли в результате качество. Я готов поручиться за качество продукта, создаваемого такими хорошими работниками. Мне всегда казалось, что высокое качество продукта в любой области в основном зависит от качества людей, создающих этот продукт. Почему же не распространить это соображение и на программное обеспечение?

---

[1] Фредерик Брукс «Мифический человеко-месяц или как создаются программные системы». – Пер. с англ. – СПб.: Символ-Плюс, 2000.

5.6. Оценка результатов применения модели зрелости процесса (CMM)

В предыдущих разделах мы задавали вопрос о том, какую ценность представляет собой ориентированная на процесс модель зрелости CMM. Мы описали начальные шаги Института программирования, направленные на то, чтобы преодолеть проблему сложности оценочных исследований преимуществ CMM.

Сначала SEI опирался на мнение экспертов, определивших содержание CMM – Уоттс Хамфри с коллегами, первоначально разрабатывавшие эти подходы в IBM и прочих местах, были теми гуру, на чьих взглядах, мнениях и опыте основывалась CMM. Вскоре после этого стали привлекать экспертов для рецензирования, и в SEI появилось много экспертов по программному инжинирингу, которые разбирали предлагаемые процессы, доводили их до ума и предлагали новые, улучшенные версии. Если объективное количественное исследование CMM было невозможно, то такой путь был лучшим из возможных. И он гораздо лучше попыток усовершенствовать наши методы создания программного обеспечения, предпринимаемых большинством наших методистов и гуру. Как отмечено в ряде мест данной книги, малочисленность количественных исследований – вероятно, крупнейшее упущение теоретических компьютерных наук и программирования.

Шло время, и исследователи в конце концов начали преодолевать этот недостаток. Ларри Путнэм (Larry Putnam) изучил эффект от перехода разрабатывающей ПО организации с уровня 1 на уровень 3 (от хаоса к среднему уровню порядка) и выявил сокращение сроков разработки, максимальной численности работников и объема работ по сравнению с исходными, составившими 170, 320 и 570% соответственно [Putnam 1994]. Это были впечатляющие цифры, которые можно было приводить как свидетельство едва ли не революционных результатов применения CMM.

К нашему счастью, эту эстафету приняли другие исследователи. Джим Херблсеб (Jim Herblseb) с рядом коллег из SEI обследовали 13 компаний, где была введена CMM, и обнаружили рост производительности на 9–67% и выявления дефектов на 6–25% [Herbsleb и др. 1994]. Хорошие показатели, но не такие сенсационные, как обнаруженные Путнэмом, и вызывающие вопросы относительно причин такой разницы в результатах.

Появились публикации о других исследованиях. Томас Хейли (Thomas Haley) провел исследование в Raytheon [Haley 1996] и выявил такие достижения:

- рост продуктивности за 8 лет на 190% (24% в год)
- сокращение числа ошибок на 1000 строк кода с 17 до 4 за 8 лет (улучшение на 53% в год)

И снова хорошие результаты, но не согласующиеся с двумя предыдущими.

Спустя несколько лет Кларк опубликовал исследование [Clark 2000], согласно которому подъем на каждый следующий уровень зрелости процесса сокращает затраты на разработку на 4–11%. Революция? Ничего похожего. Хотя и здесь – вполне приличное улучшение.

Наконец, есть еще одно, довольно необычное исследование, на которое стоит обратить внимание. В журнале, посвященном информационным системам [Harter и др. 2000], были опубликованы данные о том, что рост степени зрелости на 1% влечет сокращение затрат на разработку на 0,61% и рост качества продукта (по количеству ошибок) на 1,6%. Странность этого исследования в том, что не вполне понятно, что означает рост степени CMM на 1%. Как известно, CMM состоит из отдельных дискретных уровней – один, два, три, четыре, пять – и между ними фактически нет промежуточных. Конечно, есть ключевые процедуры, с помощью которых определяется, что организация перешла с одного дискретного уровня на другой, но в описании CMM нет ничего, что позволило бы сколь-нибудь разумно толковать смысл дробных уровней. Дело не в том, что слишком трудно понять смысл полученных результатов, а в том, что не вполне понятно, как проводивший исследование мог их получить.

Полезно вернуться к этим результатам и посмотреть на них еще раз. При этом выявляется пара обстоятельств. Во-первых, похоже, не вызывает сомнений, что методы процессов CMM приносят пользу разработке программного продукта. Эта польза, как мы видим, может быть измерена в отношении продуктивности и качества (во всяком случае, если качество измерять количеством ошибок в программе).

Второе, что обращает на себя внимание, это сложность определения точного размера этой пользы. Имеем ли мы дело с порядком величин, как хочет убедить нас Путнэм, или с небольшими процентами, вплоть до незначительных 4%, как склонны показывать другие исследования?

Третье, что бросается в глаза (скорее, постепенно проявляется, и то лишь если взглянуть на приведенный ниже перечень ссылок!), это публикация таких исследований исключительно в журналах, посвященных практическому программированию. А где количественные оценки пользы CMM из академических журналов?

Теперь вернемся еще дальше назад к выводам относительно этих выводов. Мы обнаруживаем типичный вариант выполнения оценочного исследования в мире программирования. Оказывается, что их явно мало,

а те, что проведены, редко публикуются в научной литературе. Кроме того, часто выявляется огромная вариабельность результатов проведенных исследований. Технология X лучше технологии Y, как показывают эти исследования, и разница может быть любой в диапазоне от -5% до +112%. Не очень-то радует. В итоге некоторые исследователи отказываются от оценочных исследований – как ввиду трудности последующей публикации в научных журналах и, следовательно, бесполезности для карьеры, так и по причине сложности проведения и сравнительной неубедительности!

Но мы не должны сдаваться. Потому что без таких исследований нам всем придется полагаться на мнение неких экспертов/гуру по поводу достоинств технологий, в продаже которых, как часто бывает, сами эти эксперты заинтересованы. Да, они самым дотошным образом проанализировали эти технологии, но никогда не проверяли в практических условиях, с применением объективных и сколь-нибудь точных методов исследований.

Поэтому я хочу завершить этот раздел словами благодарности в адрес исследователей, давших нам приведенные выше цифры – пусть даже они столь сильно разнятся. Мечтаю о том времени, когда наша сфера деятельности созреет, мы лучше овладеем методами исследований и станем получать близкие и непротиворечивые результаты количественных исследований. Вот тогда и начнется действительный прогресс в области программирования.

Ссылки

CLARK 2000 – «Quantifying the Effects of Process Improvement on Effort», *IEEE Software*, Nov., 2000; Bradford K. Clark.

HALEY 1996 – «Software Process Improvements at Raytheon», *IEEE Software*, Nov., 1996; Thomas J. Haley.

HARTER ET AL 2000 – «Effects of Process Maturity on Quality, Cycle Time, and Effort in Product Development», *Management Science*, April 2000; Donald E. Harter, Maryuram S. Krishnan, and Sandra A. Slaughter.

HERBSLEB ET AL 1994 – «Software Process Improvement – State of the Payoff», American Programmer, Sept., 1994; J. Herbsleb, D. Dubrow, J. Siegel, J. Ruzum and A. Carleton.

PUTNAM 1994 – «The Economic Value of Moving up the SEI Scale», *Managing System Development* 14 (7): 1-6, 1994; L.H. Putnam.

5.7. Когда лучше ориентироваться на продукт, а когда – на процесс?

Есть люди процесса, а есть люди продукта. По-видимому, это врожденное различие между людьми.

Один мой давний знакомый любил воздушных змеев. Он любил их делать. Запускать их ему было не очень интересно. Он обожал процесс, а не продукт. Он делал очень красивых змеев – лучше я не встречал. Несомненно, они побеждали бы в конкурсах на самого красивого змея. Сомневаюсь, чтобы они победили в конкурсе на самого летучего змея, но, что характерно, моего знакомого это не заботило.

Большинство из нас в своем профессиональном или любительском деле не может позволить себе такую же роскошь, как мой знакомый. Скажите своему боссу, что ваша новая программа победит в конкурсе на самого красивого змея, но не займет призового места в конкурсе на самого летучего змея, – и он сам отправит вас в полет! Когда речь идет о работе, процесс, в основном, лишь средство создания продукта.

Поэтому важно определять, когда нужно сосредоточить внимание на продукте, а когда – на процессе. Думаю, то и другое в какой-то момент заслуживает особого внимания, о чем и хочу поговорить более конкретно. Будем исходить из того, что мы, как правило, должны заниматься продуктом, а процессом интересоваться тогда, когда не удается как следует разглядеть продукт.

Что я имею в виду? Возьмем, к примеру, процедуру заключения контракта на разработку программы. Я хочу составить контракт, который станет основой вашей работы по созданию программного продукта для меня. Я хочу, чтобы этот продукт соответствовал всем требованиям качества, выполнял нужные мне функции, удовлетворял пользователей и был изготовлен с соблюдением установленных сроков и бюджета.

Моя задача при составлении контракта – максимизировать вероятность того, что все так и произойдет. Казалось бы, это просто, но на деле это не так. Многое из того, что я хочу получить в этом продукте, трудно сформулировать юридически обязывающим образом. Например, один из элементов качества, который я хотел бы видеть в продукте, это высокая степень модульности, которая позволит вам – или тому, кто станет сопровождать продукт в будущем, – легко его модифицировать. Но как описать, что код должен быть модульным? Описание должно быть таким, чтобы, случись судебное разбирательство, было совершенно ясно, чего я требовал, и если вы этого не сделали, суд согласится с тем, что я не получил того, что мне полагалось.

Задумайтесь над этой непростой проблемой. Модульность, один из главных компонентов наших представлений о хороших программах, очень трудно определить юридически обязывающим образом. Прежде всего, придется сформулировать в контракте, что такое модуль. Попробуйте это сделать сами! Теперь попробуйте еще раз, но со связанными руками, чтобы вы не могли ими размахивать! О чем бы вы ни говорили – о функциональном сцеплении, слабом связывании, единственности точек входа и прочих вещах, которыми обычно описывается высокая степень модульности, – вы постоянно сталкиваетесь с тем, что один с трудом определяемый термин порождает другой такой же.

Тот, кто пытался определить в контракте, что собой представляет хорошая модульность, в итоге часто делал это очень неудачно. Требование должно быть таким, чтобы было очевидно, выполнено оно или нет, а в результате часто определяют модульность как количество сгруппированных строк кода. Например, «программа должна быть разбита на модули, каждый из которых выполняет одну или очень малое число функций и содержит не более 50 строк кода». Есть множество программных спецификаций с подобными определениями модулей в контракте, особенно для министерства обороны США.

И кто это сказал, что в хорошем модуле должно быть 50 (или 100, или 200) строк кода? Пытаясь дать количественное определение, которое можно проверить, мы скатились к абсурду. На самом деле, есть исследование [Card 1990], автор которого показывает, что количественные ограничения размера модуля часто приводят к появлению плохих модулей.

Что остается составителю контракта? Если нельзя определить продукт, определите процесс. Выберите и зафиксируйте ряд контрольных точек в разработке продукта, в которых, среди прочего, можно проверить соответствие не столь строгому, но более содержательному определению модуля. Это решение старого парадокса: пусть я не могу описать, какой должна быть хорошая картина, но я сразу узнаю ее, когда увижу. Благодаря процессу мы можем оценить качество на ранних стадиях.

Вот *эту* роль и играет, по моему мнению, процесс в условиях ориентированности на продукт. Если не можете определить продукт с желательной вам степенью ясности, то дайте ясное определение процесса получения этого продукта. Задайте контрольные точки. Определите промежуточные продукты, которые должны быть готовы или продемонстрированы в этих точках. Постарайтесь, чтобы эти промежуточные продукты действительно характеризовали продвижение к конечному продукту. Оценивайте промежуточные продукты исходя из этого критерия.

Далее, если выяснится, что есть процесс, применение которого оказывается разумным при разработке трудно специфицируемых продуктов,

можно попытаться его формализовать. Впредь при создании продуктов этого типа станем применять этот процесс. Результат его формализации можно назвать методологией. Обратите внимание на то, как мы пришли к определению методологии, двигаясь в восходящем направлении: перед нами стояла задача (продукт, который трудно специфицировать), мы указали метод ее решения (определить процесс, который поможет получить нужный продукт), оценили этот метод (воспользовавшись им несколько раз) и только потом определили методологию (формализовав применяемый процесс).

(Такое отклонение в сторону обсуждения методологии может показаться неожиданным. Но сколько существует методологий, разработанных по нисходящему принципу разными гуру – или даже вчерашними студентами, – не написавшими ни одной строчки кода за последние 15 лет (или за всю жизнь!). Тот, кто не разработал ни одного продукта согласно некоторому процессу, чтобы оценить это процесс, *не должен* предлагать никаких методологий.)

После какого-то периода применения этого процесса-методологии можно даже задуматься о формализации более высокого уровня (например, о статистическом контроле процесса). При использовании этого процесса обычно происходят такие-то вещи. Если они не происходят, это сигнал тревоги: при разработке данного продукта процесс работает не так, как обычно. Возможно, здесь есть какая-то проблема, но необязательно. Тем не менее, это сигнал о том, что необходимо расследование. Таким образом, статистический контроль процесса оказывается полезным инструментом для выявления отклонений от нормы.

Прежде чем мы завершим обсуждение этой темы, нужно сделать еще одно важное замечание. Хороший процесс не *гарантирует* получение хорошего продукта. Это не то же самое, что «хороший процесс не приводит к хорошему продукту»: интуиция говорит, что приводит. Однако с тех пор, как первый гуру изобрел первую методологию (формализованный процесс, так?), вошло в моду заявлять, что применение «моей» методологии гарантирует хороший результат. Это не так, и тому есть доказательства. Том Демарко, автор многих замечательных книг и статей, несколько лет назад сообщил на международной конференции по программному инжинирингу, что им был изучен ряд продуктов, созданных специально подготовленными людьми с применением одной и той же методологии, и между этими продуктами обнаружилось весьма мало сходства. А если взять пример из давней истории, то претензии на то, что структурное программирование приводит к появлению кода, не содержащего ошибки, было опровергнуто несколько лет назад клиентами из «New York Times», заказавшими первый крупный проект с применением структурного программирования, мнение которых: «Может, структурное программирование

и сокращает количество ошибок до одной на тысячу строк, но исполнитель должен нам еще порядка миллиона строк». Иными словами, заявления о надежности продуктов, созданных с применением структурного программирования, сильно приукрашивали положение дел в те времена.

Но спустимся на землю. Цель данного раздела – показать, что в одних ситуациях нужно сосредоточиваться на продукте, а в других – на процессе. Конечный продукт, по крайней мере в обычных условиях, всегда должен стоять во главе угла. Перенос внимания на процесс, когда продукт не очень четко очерчен, является вынужденным. Но если уж заниматься процессом, то всерьез. И если удалось определить процесс, оказавшийся успешным, можно применить к нему ряд интересных формализующих подходов.

Ссылки

Card 1990 – *Measuring Software Design Quality,* Prentice-Hall, 1990; David N. Card with Robert L. Glass.

Глава 6 | Интеллектуальные и канцелярские задачи

Введение

6.1. Программы: трудно или легко их делать?

6.2. Задачи программирования: интеллектуальные, канцелярские или... творческие?

6.3. Почему люди «покупаются» на революционные идеи

Введение

Эффективный симбиоз человека и компьютера состоит в том, что непрограммируемые задачи решает человек, а программируемые – компьютер. Непрограммируемая область включает в себя целостную, глобальную деятельность, в том числе эвристики, ценности, позиции, эмоции и юмор, а программируемая – подробные последовательные пошаговые алгоритмы.

Моше Рубинштейн

«Tools for Thinking and Problem-Solving»,
Prentice-Hall, 1986

Позвольте рассказать вам одну историю. Она описывает эволюцию некоторой научной статьи. (Читайте-читайте, не пожалеете!) Это рассказ об исследовании с целью ответить на вопрос: в какой мере программирование является интеллектуальной деятельностью?

Зачем это нужно? Разве не очевидно, что программирование – сложная, требующая умственных усилий работа?

К сожалению, большинство ответит на этот вопрос отрицательно. Повсюду мы встречаем свидетельства того, что люди думают именно так. Руководителей удивляет количество времени, необходимое программистам для «простого» изменения в программе. В учебных заведениях пренебрегают программистскими курсами, порой исключая их из расписания по причине «отсутствия в них интеллектуального содержания». (Не смейтесь – такое бывало в нескольких первоклассных учебных заведениях всего лишь лет десять назад и не исключено, что может повториться сегодня!) Разные гуру возвещают, что последняя модная новинка устранит потребность в программистах. Исследователи ищут финансовую поддержку для проектов, ставящих целью «автоматизацию» процесса создания программного обеспечения. Правительственные организации ищут подрядчиков, которые разработают им методы, позволяющие «на порядок» улучшить наши возможности по созданию и/или сопровождению программ. И это далеко не всё. Так или иначе, все эти люди уверены в том, что создавать программное обеспечение и работать с ним легко.

Эти ошибочные представления нанесли огромный ущерб компьютерной области. Аутсорсинг[1] – их результат. Закрытие учебных программ по программированию – тоже их результат. На их же счет следует отнести недостаточное финансирование жизненно важных (вместо отвлеченных) исследований в программировании, абсурдные требования к сокращению стоимости и длительности разработок, утрату доверия.

Но вернемся к моему рассказу. Какое-то время я был озадачен таким странным несоответствием. Я размышлял не только о том, откуда у людей столь странные представления, но и – подсознательно – о том, как бы их переубедить. Возможно, здесь помогло бы хорошо подготовленное исследование на эту тему. Но какое?

И тогда мне попались эти кассеты. Что за кассеты? Мои коллеги-исследователи занимались протокольным анализом методик разработки программного обеспечения. Для этого они снимали на видео начинающих системных аналитиков в процессе решения задачи. По какой-то причине, которую уже не могу вспомнить, однажды я вставил одну из кассет в плеер и из чистого любопытства стал смотреть.

Глядя на экран, я поразился тому, что большую часть времени аналитики не делали абсолютно ничего! Смотреть на это было ужасно скучно. И при этом я понимал: поскольку объекты съемки знали, что их действия фиксируются и хронометрируются, они усиленно занимаются решением задачи.

И тут до меня дошло. Когда я скучал, наблюдая на экране абсолютное бездействие объекта съемки, все это время он думал. А то время, когда объект регистрировал на бумаге результаты своих размышлений, напротив, было отдано канцелярской работе. Там, на кассете, была живая иллюстрация к вопросу о том, является ли программирование сложной интеллектуальной работой. А что если измерить, сколько времени каждый объект размышлял, а сколько – записывал и рисовал?

Именно это мы (я и мои коллеги по исследованию) и сделали. Мы выяснили, сколько времени каждый из нескольких объектов съемки тратит на умственную и на канцелярскую работу, придумали другую систему, чтобы измерить такого же характера явления, но иным способом – чтобы быть уверенными в устойчивости наших результатов, а затем написали статью и послали ее в хороший журнал.

---

[1] Аутсорсинг – передача заказов сторонним организациям и в развивающиеся страны. – *Прим. перев.*

И тут нас постигло разочарование. Исследуя безусловно отличную тему, мы применили довольно посредственные методы. Иными словами, в основной идее исследования был ряд изъянов.

1. К примеру, всегда ли бездействие совпадало с умственной работой?
2. Состояла ли активность только в канцелярской работе?
3. Стоило ли выбирать для наблюдения только начинающих?
4. Стоило ли измерять те показатели, которые нас интересовали, только на простых заданиях?

Все эти законные вопросы, как мы и ожидали, были заданы рецензентами того журнала, в который мы направили статью. Мы надеялись, что важность изучаемого вопроса перевесит недостатки методологии исследования. К сожалению, вышло иначе. Статья была отклонена.

Мы направили статью в другой журнал, и на сей раз – если опустить детали – исход оказался более благоприятным. Статья была принята и по прошествии должного времени опубликована.

А затем случилось самое интересное. В популярном журнале компьютерных новостей «InformationWeek» появилась короткая заметка о выводах, содержащихся в нашей статье. И сразу же посыпались электронные письма и телефонные звонки. В течение одного-двух месяцев пришло более ста таких писем от тех, кто заинтересовался результатами исследования и хотел узнать о них подробнее. Поток писем иссяк лишь много месяцев спустя – сейчас, когда я пишу этот текст. Каким бы ни было качество исследования, его тема задела компьютерный мир за живое.

Та статья, как вы можете теперь догадаться, включена в эту книгу и следует за введением в эту главу. Несмотря на спорный метод исследования, ее выводы оказались на удивление последовательными. Интеллектуальные аспекты преобладают над канцелярскими с огромным отрывом. (С каким именно, сказано ниже.) Хотя нужны дополнительные исследования, нет сомнений – в настоящий момент, – что те, кто считает программирование легкой работой, явно не правы.

Теперь вы знаете всю историю эволюции одного исследовательского проекта, конец которой был счастливым.

(P. S. Еще одно замечание. Когда результаты сравнения «интеллектуальность или канцелярия» были опубликованы, с нами связался еще один исследователь, интересующийся процессом творчества, покойный профессор Дэн Кугер (Dan Couger) из Университета Колорадо, и предложил продолжить работу, исследовав то, в какой мере программирование является не просто интеллектуальной, но творческой работой. (Что могло быть ближе книге о творчестве?!) С его содействием – он дал нам хорошее определение того, что такое творчество, – мы провели новое исследова-

ние. Полученные результаты, хотя и не столь захватывающие, как в первой статье, также представлены в этой главе.)

Во всех других введениях к главам этой части я определял основные используемые термины. Прежде чем двигаться дальше, определим «интеллектуальный» как относящийся к интеллекту – «способности разума к рассуждению и приобретению знаний (в противоположность чувствам и инстинктам)», и «канцелярский» (clerical) – «свойственный клеркам, связанный с копированием или записыванием чего-либо».

Теперь можно перейти к сравнительному изучению интеллектуального и канцелярского (а также творческого) аспектов разработки программ.

6.1. Программы: трудно или легко их делать?

Авторы статьи – Роберт Л. Гласс и Айрис Весси (Iris Vessey).

Представления о сложности создания программного обеспечения противоречивы. С одной стороны, благодаря многочисленным профессиональным журналам и множеству учебных курсов, посвященных различным аспектам программирования, а также высказываниям именитых компьютерных ученых, таких как Дэвид Парнас и Дональд Кнут, есть прочная уверенность в том, что «писать программы трудно» [Parnas 1986, Knuth 1989].

С другой стороны, распространена точка зрения, что писать программы легко. Вот примеры из [MBA 1990]:

> «Проще научить бизнесмена технологии, чем технаря – искусству управления».
>
> «Руководить АСУ должен бизнесмен широкого плана, а не узкий технический специалист. Я мог бы научить программированию даже свою матушку».

Более того, ореол, возникший в 1990-х вокруг автоматизированных рабочих мест CASE, появился во многом благодаря представлению, что значительную часть работы по созданию программ можно автоматизировать. Возьмите, например, такое высказывание [McClure 1988]:

> «Рабочее место оснащено интегрированными инструментами, позволяющими целиком автоматизировать разработку и сопровождение программных систем и управление программными проектами. Результат, полученный на одном этапе жизненного цикла, автоматически передается на следующий этап; окончательный продукт – работающая программная система с документацией».

Эта цитата должна убедить в том, что в инструменты разработки программ могут заменить человека.

Очевидно, что эти точки зрения диаметрально противоположны. Согласно одной из них, программировать нелегко даже тонким и изощренным умам. Согласно другой, программировать настолько просто, что этим может заниматься едва ли не каждый. Где в этих представлениях ошибка? Может быть, верны обе точки зрения? Если нет, то какая из них ошибочна?

Конечно, сложность программы тесно связана со сложностью задачи, которую должна решать эта программа. Поэтому вполне программы могут быть как элементарно простыми, так и невероятно сложными. Но все же мы отклонились от поставленного выше важного вопроса: насколько трудна работа программиста?

Ответ можно дать разными способами. Опыт Парнаса с проектом А-7 показывает, что создание программного обеспечения – действительно сложная задача [см., к примеру, Parnas 1985]. Что касается автоматизированных методов изготовления программ, то Рич и Уотерс ссылаются на досужий миф об автоматическом программировании [Rich 1988].

Эмпирическое свидетельство приводится в [Fjelstad 1979], где показано, что при сопровождении программ 45% времени тратится на то, чтобы понять, какое изменение нужно сделать и как работает изменяемая программа, 35% времени уходит на проверку сделанной модификации и только 20% – на собственно внесение изменений. Фактически 80% времени тратится на обдумывание проблемы и ее решения. Еще одно эмпирическое свидетельство [Woodfield 1979] показывает, что на каждые 25% увеличения сложности решаемой задачи сложность программы, решающей эту задачу, вырастает на 100%. Эти данные указывают на то, что сложность программного обеспечения – это проблема масштабирования: простые задачи решить легко, сложные – очень трудно.

Но других фактических материалов мало. Тот же вопрос встречается в заявлениях вроде «...тестирование состоит из двух этапов – планирования и исполнения. В то время как на втором этапе можно эффективно применять средства автоматизации, планирование по большей части основывается на человеческой изобретательности и компетентности» [Bertolino 1991]. Найти конкретные или количественные ответы трудно.

Но найти их – важно. Нынешнее положение свидетельствует о широком распространении, если не о господстве, той точки зрения, что разрабатывать программы и управлять программными проектами может каждый.

Если программирование – главным образом канцелярская деятельность, то такая точка зрения обоснованна. В противном случае она вредна. Та-

ким образом, крайне важно выяснить соотношение интеллектуального и канцелярского в программировании.

В данной работе мы делаем начальные исследовательские шаги с целью получения ответа на этот вопрос. Представлены результаты двойственного подхода к изучению интеллектуального и канцелярского компонентов процесса создания программного обеспечения. В следующем разделе мы рассмотрим задачи, решаемые при разработке программ: какие из них требуют интеллектуальных усилий, а какие являются по большей части канцелярскими. Затем мы представим результаты эмпирического исследования, в ходе которого регистрировалось, сколько времени тратится при решении задачи системного анализа на обдумывание и рисование диаграмм соответственно. В конце мы представим выводы и следствия из проведенного исследования.

Классификация задач программирования

В табл. 1 и 2 систематизируются задачи программирования. В табл. 1 перечислены задачи, решаемые CASE-инструментами, как это описывается в литературе, посвященной CASE. Поскольку эти задачи автоматизированы, можно, по-видимому, считать, что они носят в некоторой степени более канцелярский характер, чем не автоматизированные. В табл. 2 задачи представлены более абстрактно, вне зависимости от того, есть ли CASE-инструменты для их решения.

Чтобы выяснить, в какой мере программирование является интеллектуальной, а в какой канцелярской работой, можно было бы попросту выбрать в этих списках интеллектуальные и канцелярские задачи, сделав вывод вроде «X процентов задач программирования являются интеллектуальными, а Y процентов – канцелярскими». Мы бы по-прежнему не знали, какую часть общей работы над программным продуктом составляет та или иная категория, но все же это был бы существенный шаг вперед в понимании проблемы, даже в отсутствие количественной оценки.

Однако этому мешает ряд обстоятельств. Во-первых, таблицы показывают, что перечни задач плохо согласуются между собой – во всяком случае, на уровне формулировок. Во-вторых, тому, кто не знаком с программированием, далеко не очевидно, какие задачи являются интеллектуальными, а какие канцелярскими. В третьих, в отсутствие конкретного контекста даже среди специалистов по программированию могут возникнуть разногласия по поводу классификации задач.

Чтобы преодолеть эти трудности, к проведению нужной классификации мы подошли просто. Мы предложили двум экспертам-профессионалам в области программирования независимо оценить каждую задачу как интеллектуальную (и) или канцелярскую (к). Те задачи, которые имеют

признаки обеих категорий, были отнесены к категории неопределенных (н). Эти два эксперта высказали одинаковое мнение в 77% случаев. Разногласия были разрешены с помощью третьего эксперта, голос которого в спорных случаях был решающим. Результаты этой процедуры представлены в правых столбцах табл. 1 и 2.

Таблица 1. Задачи, решаемые CASE-инструментами[a]

Задача программирования[b]	**Код**
Функции представления	
Представление проекта	и
Конструирование моделей	и
Выбор языка и соглашений, применяемых в представлении	и
Представление взаимосвязей	и
Объединение сущностей или процессов	н
* Отображение атрибутов объекта	
Сопровождение описаний	к
Выработка соглашений по именованию	и
Поддержка единого определения	к
* Переход к другой модели	
Перерисовка диаграммы	к
Отображение в функциональное описание	н
Объединение эквивалентных процессов	н
* Одновременный показ нескольких видов	
Выбор модели	и
Функции анализа	
Проверка непротиворечивости модели	и
Поиск эквивалентных структур	и
Поиск ненужных или лишних связей	н
Обнаружение противоречий	н

[a] Henderson, J. C. and Cooprider, J. G. «Dimensions of IS Planning and Design Technology», Information Systems Research, Vol. 1, No. 3, 1990, 227–254.

[b] Задачи, основанные на технологии.

* Задачи, основанные на технологии, не рассматриваемые в данном анализе задач.

Задача программирования[b]	**Код**
Определение влияния изменений в проекте	и
Поиск в проекте сходных объектов	и
Предложение решений	и
Оценка характеристик	и
Поиск в проекте заданных характеристик	и
Моделирование производственных условий	и
Выявление нарушений правил	к
Отслеживание взаимосвязей	н
Выявление различий	и
Рекомендация общей модели	и
* Выполнение операции над частью проекта	
Функции преобразования	
Генерация исполняемого кода	и
Преобразование спецификации	и
Преобразование представления	и
Составление документации	н
Обратное проектирование	и
Создание макетов экранов	и
* Импорт/экспорт данных	
Создание макетов задач и конечных продуктов	и
Распространение модификаций	к
Функции управления	
Определение рецензентов	н
Предоставление административной информации	н
Учет ответственности	к
Учет модификаций	к
Предоставление административной информации по нескольким проектам	н
Определение допущенных к модификациям	и
* Замораживание части проекта	
Управление контролем качества	и
Изменение правил	и

Таблица 1 (продолжение)

Задача программирования[b]	**Код**
Помощь в установке приоритетов	и
Определение сроков для задач/проектов	и
Напоминание исполнителям о сроках сдачи	к
Слияние версий	к
Выработка метрик	и
Ведение списка требований и степени их реализации	к
* Временная приостановка задачи с целью продолжить работу	
Функции совместной деятельности	
* Переговоры с другими пользователями инструментов	
* Обеспечение одновременной работы группы над задачей	
* Отправка сообщений другим пользователям инструментов	
* Обеспечение параллельного доступа к словарям/диаграммам/пр.	
* Обеспечение взаимодействия групп (коллективного обсуждения)	
* Создание электронных примечаний	
* Обеспечение анонимной критики	
Извещение проектировщика об изменениях	к
Составление каталога макросов	к
Содействие альтернативным проектам	и
Функции поддержки	
Создание кратких справочников	и
Создание учебных материалов	и
Определение внешних источников информации	и
Создание шаблонов/примеров для учебников/демонстрационных материалов	и
Поиск в других сегментах инструмента	и
Разъяснение сути причин противоречивости части проекта	
Ожидаемые ошибки пользователей по прежнему опыту	и
Разрешение отката нескольких команд	и
Вывод результатов на разные носители	н
Регистрация командных макросов	и
Подготовка отчетов и документов	н

Задача программирования[b]	**Код**
* Подготовка страниц с изменениями	
* Увеличение модели для показа более мелких деталей	
Создание библиотеки настроенных моделей	и
* Подготовка, редактирование, хранение, пересылка и получение документов	
Хранение версий проекта	к
* Компоновка проекта с библиотекой для тестирования	
* Разработка, выполнение и хранение настроенных отчетов	

Таблица 2 (а). Выполняемые задачи программирования[a]

Задача программирования	**Код**
Задачи программных требований	
Определение программных требований	
Анализ контекста	и
Изучение требований путем опроса	и
Вывод программных требований из системных требований	и
Анализ задач для выработки требований к интерфейсу пользователя	и
Определение ограничений	и
Анализ требований	
Оценка возможных проблем	и
Классификация требований	и
Оценка осуществимости и рисков	и
Представление требований	
Использование моделей	и
Роли для прототипирования	и
Сообщение требований	и
Подготовка к проверке (критерии, технология)	и
Управление процессом определения требований	и

[a] Brackett J. W. «Software Requirements», Software Engineering Institute, SEI-CMU-19-1.1 – 1989.

Таблица 2 (б). Выполняемые задачи программирования[a]

Задача программирования[b]	**Код**
Общие задачи программирования	
Анализ требований	и
Анализ потоков данных	и
Функциональная декомпозиция	и
Разработка проектной документации и спецификаций	и
Контроль обновления документов	к
* Доступ к проектам многократного использования	
Разработка нового кода	и
* Доступ к коду многократного использования	
Анализ и модификация имеющегося кода	и
Реструктуризация имеющегося кода	к
Удаление мертвого кода	и
Рецензирование проекта	и
Инспекция кода	и
Персональная отладка	и
Создание контрольных примеров	и
Контроль библиотеки тестирования	к
Анализ дефектов	и
Создание документации для пользователей	и
Создание электронной подсказки и учебников	и
Обучение пользователей	и

[a] Jones T. C. «Why Choose CASE?», *American Programmer*, vol, 3, No. 1 (Jan 1990), 14–21.

[b] Задачи, основанные на технологии.

* Задачи, основанные на технологии, не рассматриваемые в данном анализе задач.

В табл. 3 представлен итог данной классификации – процент интеллектуальных, канцелярских и неопределенных задач для группы задач CASE и не-CASE. Из табл. 3 явствует, что даже при использовании CASE-средств интеллектуальные задачи преобладают над канцелярскими (соответственно, 62% и 21%). Для задач не-CASE разрыв усиливается еще

больше: 83% против 17%. В целом интеллектуальные задачи преобладают в соотношении 71% против 18%.

Эта элементарная классификация показала количественное преобладание интеллектуальных задач программирования. Однако с учетом ограниченности такого подхода оценивать полученный результат следует лишь как элемент более общей картины.

Таблица 3. Классификация задач CASE и не-CASE

Категория задач	Канцелярские	Не определено	Интеллектуальные
CASE-задачи по Henderson and Cooprider (1990)	14[a] (21)	11 (17)	40 (62)
Не-CASE задачи по Brackett (1989)	0	0	13 (100)
Jones (1990)	3 (17)	0	15 (83)
Суммарно	17 (18)	11 (11)	68 (71)

[a] Числа в таблице указывают количество задач в каждой категории; в скобках указаны проценты.

Результаты экспериментов

Один из недостатков изложенных выше результатов – то, что они касаются количества задач, а не времени, потраченного на их решение. В данном разделе мы опишем другой подход к решению вопроса о соотношении интеллектуального и канцелярского. Мы изучили, сколько времени разработчики тратят на обдумывание задачи программирования и сколько – на действия по результатам этих размышлений.

В другом исследовании по системному анализу [Vessey 1992] на видео записывалась работа ряда лиц во время решения ими задач. Целью этого исследования было узнать, как системные аналитики применяют различные методологические подходы при определении информационных требований в двух предложенных им ситуациях. Объекты изучали поставленную задачу, обдумывали, как представить задачу в виде произвольной графической спецификации, а затем рисовали ее (в данном случае, от руки). По результатам видеосъемки можно было измерить, сколько времени объекты исследования тратили на обдумывание, а сколько – на рисование.

Результаты исследования представлены в табл. 4. Впечатляет устойчивость полученных результатов. Большинство объектов примерно 21% времени потратило на бумажную работу (рисование графических представлений), а оставшиеся 79% – на обдумывание. Диапазон времени на обдумывание составил от 72 до 85% и явно невелик для группы лиц, имеющих между собой мало общего по принципу отбора.

Несмотря на обращающую на себя внимание согласованность результатов исследования, важно учесть ряд присущих ему ограничений. Во-первых, отбор осуществлялся среди аспирантов с малым опытом системного анализа или вообще без него. Во-вторых, результаты получены по материалам исследования, проводившегося с другой целью: этот эксперимент не давал возможности контролировать относящиеся к делу параметры. В-третьих, задача спецификации информационных требований по своей природе является неструктурированной и потому интеллектуальной, так как ее решение связано с начальным моделированием реальной задачи.

Однако полученные результаты настолько согласуются между собой, что можно утверждать: по крайней мере, среди аспирантов интеллектуальная работа при системном анализе примерно вчетверо превышает по продолжительности работу, скорее относящуюся к канцелярской.

Таблица 4. Длительность обдумывания и записи результатов при системном анализе

Объект	Задача	Общее время (мин.)	Время на запись (мин.)	% от общего времени	
				Запись	Обдумывание
S3	1	153	23,1	15	85
	2	106	20,5	19	81
S4	1	117	24,2	21	79
	2	26	6,8	26	74
S6	1	108	18,7	17	83
	2	60	12,4	21	79
S10	1	51	11,0	22	78
	2	41	10,0	24	76
S11	1	61	12,8	21	79
	2	87	15,3	18	82
S12	1	61	16,9	28	72
	2	67	14,3	21	79

Тем не менее, остается нерешенным ряд вопросов: а) будут ли иными эти показатели для опытных практиков? б) будут ли аналогичными результаты для иных видов деятельности помимо системного анализа? в) связаны ли результаты с конкретным выбором задач в проведенном исследовании? (Можно заметить, однако, что разница между показателями для двух изучавшихся задач очень невелика.) Ответы на эти вопросы способствовали бы лучшему пониманию соотношения интеллектуальной и канцелярской деятельности при создании программного обеспечения.

Выводы и следствия

Тема соотношения интеллектуальной и канцелярской работы в процессе программирования представляется важной и довольно противоречивой. Сбор информации по этой теме помог бы ответить на такие вопросы, как:

1. На какие выгоды автоматизации программирования можно РЕАЛЬНО рассчитывать?
2. Можно ли привлекать относительно неподготовленных людей (например, необученных пользователей) к созданию и сопровождению программ?
3. Насколько важна техническая подготовка для успешного руководства программными проектами?

Данное исследование дает некоторые начальные, но отличные от общепринятых данные для решения этих важных вопросов. Вызывает интерес согласованность полученных результатов между собой. Проанализировав решаемые задачи, мы видим, что интеллектуальные задачи преобладают над канцелярскими в соотношении примерно 71% к 18%, или почти 4 к 1. В экспериментальном исследовании временых затрат при системном анализе мы наблюдаем преобладание интеллектуальных задач над канцелярскими в соотношении 79% процентов к 21%, или почти 4 к 1.

Количественное соотношение интеллектуальных и канцелярских задач удивительно близко к соотношению длительностей мыслительной и канцелярской работы при решении задач системного анализа. Данное начальное исследование показывает, что задачи программирования в значительно большей мере являются интеллектуальными, чем канцелярскими. Поэтому можно сделать предварительный вывод, что даже простейшие задачи программирования требуют больше интеллектуальных усилий, чем предполагает изложенная во введении точка зрения, согласно которой «писать программы может каждый». Разумеется, результаты этих первых шагов не дают оснований для окончательных выводов. Необходима дополнительная работа для лучшей дифференциации и исследования этих результатов.

Благодарности

Авторы признательны Умешу Бхатия (Umesh Bhatia) и Сью Конджер (Sue Conger) за помощь в этом исследовательском проекте. Статья по этим материалам была опубликована в «Journal Information and Management» (ноябрь 1992 года).

Ссылки

BERTOLINO 1991 – «An Overview of Automated Software Testing», *Journal of Systems and Software,* May 1991; A. Bertolino.

FJELSTAD 1979 – «Application Maintenance Study: Report to Our Respondents», *Proceedings of GUIDE 48*, The Guide Corporation, Philadelphia, 1979; R. K. Fjelstad and W. T. Hamlen.

KNUTH 1989 – Keynote address, World Computer Congress, 1989, as reported in *System Development*, Dec. 1989; Donald E. Knuth.

MBA 1990 – «More on MBAs in MIS», letters to the editor in response to an article on whether the employment of MBAs in Information Systems positions helps or hurts organizations, *Information Week*, March 19, 1990.

MCCLURE 1988 – «The CASE for Structured Development», *PC Tech Journal*, August 1988, 51–67; Carma McClure.

PARNAS 1985 – «The Modular Structure of Complex Systems», *IEEE Transactions on Software Engineering*, March 1985; David L. Parnas, et al.

PARNAS 1986 – Presentation at Seattle University graduate program seminar, 1986; David Lorge Parnas.

RICH 1988 – «Automatic Programming: Myths and Prospects», *Computer*, August 1988, 40–51; Charles Rich and Richard Waters.

VESSEY 1992 – «Specifying Information Requirements: Factors Influencing Problem Analysis», unpublished manuscript; Iris Vessey and Sue A. Conger.

WOODFIELD 1979 – «An Experiment on Unit Increase in Program Complexity», *IEEE Transactions on Software Engineering*, March, 1979; Scott Woodfield.

6.2. Задачи программирования: интеллектуальные, канцелярские или... творческие?

Авторы статьи – Роберт Л. Гласс и Айрис Весси.

Исследований характера работы по созданию программного обеспечения проводилось мало. У нас, таким образом, мало научных фактов в поддержку одной или нескольких из следующих точек зрения:

- Задачи программирования просты и поддаются автоматизации.
- Задачи программирования сложны и требуют умственных усилий.
- Задачи программирования в значительной мере требуют творческого подхода.

Недостаточная исследованность этих вопросов удивляет, поскольку у каждой из указанных точек зрения есть интеллектуальные и политические сторонники в компьютерном сообществе, и весьма маловероятно, чтобы все три точки зрения были верны.

Рассмотрим примеры заявлений.

- Руководители компаний, активно задействующих компьютеры, делали публичные заявления вроде «Я мог бы научить программированию даже свою матушку» [MBA 1990], выражая этим уверенность в тривиальности задач программирования.
- Гуру, занимающиеся компьютерным будущим, предполагают, что нас ждет «разработка программ без участия программистов» [Martin 1982], подразумевая, что написание программ легко может быть освоено конечными пользователями или даже «автоматизировано».
- Ведущие ученые специалисты в области программного инжиниринга считают создание программного обеспечения наиболее сложным из когда-либо существовавших видов человеческой деятельности [USNRC 1990], что свидетельствует о заведомой тщетности попыток найти средства, позволяющие на порядок поднять эффективность решения задач программирования.
- Авторы популярных книг в этой области осуждают попытки введения формальных и точных процессов разработки программного обеспечения на том основании, что подобные подходы снижают возможность творчества, необходимого при проектировании и создании программ (см., например, [DeGrace 1993]).

Широта расхождений между этими противоречивыми точками зрения и степень уверенности их сторонников делают желательным исследование задач, решаемых в программировании. Возможно ли, чтобы все эти

различные точки зрения могли каким-то образом сосуществовать? Если нет, то какая из них ближе всего к истине?

Цель данной статьи – представить результаты предшествующих исследований этих проблем и сообщить о дополнительных и несколько тревожных результатах, полученных в результате одного нового исследования.

Статья построена следующим образом. В следующем разделе описывается наше предыдущее исследование, в котором оценивалось соотношение *интеллектуального* и *канцелярского* в задачах программирования. Затем описывается новое исследование, в котором оценивался творческий характер задач. После этого анализируются результаты обоих исследований и в последнем разделе делаются выводы.

Предыдущее исследование

Один из подходов к перечисленным выше проблемам – изучить характер задач, решаемых в программировании, с целью определить, в какой мере они являются канцелярскими, интеллектуальными или творческими по своей природе.

При таком подходе сложные проблемы, о которых говорилось в предыдущем разделе, сводятся к трем другим. Если бы нам как-то удалось отнести каждую из задач программирования к одной из этих трех категорий, мы получили бы четкую, пусть даже слишком упрощенную, основу для разделения задач, приводящего к более широким ответам на интересующие нас вопросы.

В предыдущем исследовании [Glass 1992] (см. раздел 6.1) для ответа на этот вопрос авторы использовали два метода:

- Измерение времени, расходуемого начинающими системными аналитиками на обдумывание (интеллектуальная работа) и решение (в основном, канцелярская работа) задач спецификации требований для некоторых относительно простых программных проектов.
- Установление путем опроса экспертов интеллектуального или канцелярского характера задач программирования из нескольких списков, имеющихся в литературе по программированию и представляющих все этапы жизненного цикла разработки программного обеспечения.

Заметим, что эти методы – анализ временных затрат и анализ задач – относились только к части интересующей нас проблемы (соотношение интеллектуальной и канцелярской работы), не касаясь творческого характера задач программирования. Это упущение исправлено в настоящем исследовании.

Настоящее исследование

Чтобы взяться за проблему творчества, необходимы были два условия: хорошее определение творчества применительно к разработке программ и механизм оценки творческого аспекта решения задач программирования, аналогичный примененному в первом исследовании, чтобы можно было провести сравнение.

В данном исследовании используется определение творчества, основанное на работе по решению задач, представленной в [Newell 1962], и применяемое в большинстве исследований творчества в процессе создания программного обеспечения (например, [Couger 1990]). В этой работе изложена точка зрения, согласно которой решение считается творческим, если удовлетворяет одному или нескольким из следующих условий:

1. Результат обдумывания отличается *новизной* или *ценностью*.
2. Мышление отличается *нетрадиционностью*, то есть требует изменения или отказа от ранее принятых концепций.
3. Обдумывание требует *высокой мотивации* и *настойчивости*, отличаясь большой продолжительностью (непрерывно или с промежутками) или высоким напряжением.
4. Начальная постановка задачи отличается *неясностью* или *некорректностью*, поэтому часть задачи – сформулировать саму задачу.

Использованные ключевые слова «новизна», «ценность», «нетрадиционность», «высокая мотивация», «настойчивость», «неясность» и «некорректность» представляются достаточными для принятия решения о том, является ли задача программирования творческой.

Чтобы решить вторую часть задачи исследования, то есть сделать его сопоставимым с предыдущим, мы воспользовались при оценке творческого характера задач теми же списками задач программирования, что и в первый раз. К сожалению, хотя было бы желательно провести анализ времени аналогично первому исследованию, примененный тогда метод (запись протокольного анализа на видео) не позволял определить объем времени, занятый творчеством. Поэтому в данном исследовании соотношения творческого, интеллектуального и канцелярскогго мы классифицировали задачи программирования, отнеся их к одной или нескольким из этих категорий, но не вели учет времени, в течение которого объекты занимались творчеством, а не интеллектуальной или канцелярской работой.

Затем, как и в первом исследовании, мы привлекли экспертов, которые решали, какие из задач являются творческими. В предыдущем исследовании участвовали два эксперта, а в тех случаях, когда их мнения расходились, спор решал третий эксперт. В настоящем исследовании, ожидая, что принятие решений окажется более сложным, мы привлекли четырех

экспертов. Один эксперт обладал опытом как в теории, так и в практике работы с информационными системами. Второй был преимущественно ученым специалистом по информационным системам. Третий являлся ученым специалистом по программированию, а четвертый имел опыт как в теории, так и в практике программирования. Таким образом, в подготовке участников исследования наблюдалось достаточное разнообразие.

Результаты исследования представлены в табл. 1–3. (Обратите внимание, что в эти таблицы включены также результаты предыдущего исследования.) В табл. 1 и 2 представлены необработанные данные – коды, присвоенные каждой из задач экспертами, а в табл. 3 суммируются результаты из табл. 1 и 2. В табл. 1 и 2 к данным предыдущего исследования о соотношении интеллектуального и канцелярского добавлен еще один столбец, где отмечено мнение экспертов о творческом характере задачи. Каждый эксперт, считавший задачу творческой, добавлял к ее оценке четверть балла. Если все четыре эксперта соглашались в том, что задача творческая, она получала суммарную оценку в 1 балл.

Таблица 1. Задачи, решаемые CASE-инструментами[a]

Задача программирования	**Код интелл./ канц.**[b]	**Оценка творчества**
Функции представления		
Представление проекта	и	0,25
Конструирование моделей	и	0,75
Выбор языка и соглашений, применяемых в представлении	и	0,50
Представление взаимосвязей	и	0,25
Объединение сущностей или процессов	н	0,25
* Отображение атрибутов объекта		
Сопровождение описаний	к	
Выработка соглашений по именованию	и	
Поддержка единого определения	к	

[a] Henderson J. C. and Cooprider J. G. «Dimensions of IS Planning and Design Technology», Information Systems Research, 1 (3), 1990, 227–254.

[b] и = интеллектуальная задача,
к = канцелярская задача,
н = неопределенная задача.

* Задачи, основанные на технологии, не рассматриваемые в данном анализе задач.

Задача программирования	Код интелл./ канц.[b]	Оценка творчества
* Переход к другой модели		
Перерисовка диаграммы	к	
Отображение в функциональное описание	н	0,25
Объединение эквивалентных процессов	н	0,25
* Одновременный показ нескольких видов		
Выбор модели	и	0,25
Функции анализа		
Проверка непротиворечивости модели	и	0,25
Поиск эквивалентных структур	и	
Поиск ненужных или лишних связей	н	
Обнаружение противоречий	н	
Определение влияния изменений в проекте	и	0,25
Поиск в проекте сходных объектов	и	
Предложение решений	и	0,50
Оценка характеристик	и	0,25
Поиск в проекте заданных характеристик	и	
Моделирование производственных условий	и	0,25
Выявление нарушений правил	к	
Отслеживание взаимосвязей	н	
Выявление различий	и	
Рекомендация общей модели	и	0,75
* Выполнение операции над частью проекта		
Функции преобразования		
Генерация исполняемого кода	и	
Преобразование спецификации	и	
Преобразование представления	и	
Составление документации	н	
Обратное проектирование	и	0,50
Создание макетов экранов	и	0,25
* Импорт/экспорт данных		
Создание макетов задач и конечных продуктов	и	

Таблица 1 (продолжение)

Задача программирования	Код интелл./ канц.[b]	Оценка творчества
Распространение модификаций	к	
Функции управления		
Определение рецензентов	н	
Предоставление административной информации	н	
Учет ответственности	к	
Учет модификаций	к	
Предоставление административной информации по нескольким проектам		
Определение допущенных к модификациям		
* Замораживание части проекта	и	
Управление контролем качества		
Изменение правил	и	
Помощь в установке приоритетов	и	0,25
Определение сроков для задач/проектов	и	0,25
Напоминание исполнителям о сроках сдачи	и	
Слияние версий	к	
Выработка метрик	к	
Ведение списка требований и степени их реализации	и	
* Временная приостановка задачи с целью продолжить работу	к	
Функции совместной деятельности		
* Переговоры с другими пользователями инструментов		
* Обеспечение одновременной работы группы над задачей		
* Отправка сообщений другим пользователям инструментов		
* Обеспечение параллельного доступа к словарям/диаграммам/пр.		

Задача программирования	Код интелл./ канц.[b]	Оценка творчества
* Обеспечение взаимодействия групп (коллективного обсуждения)		
* Создание электронных примечаний		
* Обеспечение анонимной критики	к	
Извещение проектировщика об изменениях	к	
Составление каталога макросов	и	
Содействие альтернативным проектам	к	
Функции поддержки	к	
Создание кратких справочников	и	0,50
Создание учебных материалов		
Определение внешних источников информации	и	
Создание шаблонов/примеров для учебников/ демонстрационных материалов	и	
Поиск в других сегментах инструмента	и	
Разъяснение сути причин противоречивости части проекта	и	
Ожидаемые ошибки пользователей по прежнему опыту	и	
Разрешение отката серии команд	и	
Вывод результатов на разные носители	и	
Регистрация командных макросов	и	0,25
Подготовка отчетов и документов	н	
* Подготовка страниц с изменениями		
* Увеличение модели для показа более мелких деталей		
Создание библиотеки настроенных моделей		
* Подготовка, редактирование, хранение, пересылка и получение документов		
Хранение версий проекта		0,25
* Компоновка проекта с библиотекой для тестирования		
* Разработка, выполнение и хранение кастомизированных отчетов	к	

Таблица 2 (а). Выполняемые задачи программирования[a]

Задача программирования	**Код интелл./ канц.**[b]	**Оценка творчества**
Задачи программных требований		
Определение программных требований		
Анализ контекста	и	0,75
Изучение требований путем опроса	и	0,50
Вывод программных требований из системных требований	и	0,25
Анализ задач для выработки требований к интерфейсу пользователя	и	0,25
Определение ограничений	и	0,50
Анализ требований		
Оценка возможных проблем	и	0,75
Классификация требований	и	0,25
Оценка осуществимости и рисков	и	0,75
Представление требований		
Использование моделей	и	0,50
Роли для прототипирования	и	0,25
Сообщение требований	и	
Подготовка к проверке (критерии, технология)	и	
Управление процессом определения требований	и	

[a] Brackett J. W. «Software Requirements», *Software Engineering Institute*, SEI-CMU-19-1.1, 1989.

[b] и = интеллектуальная задача,
к = канцелярская задача,
н = неопределенная задача.

Таблица 2 (б). Выполняемые задачи программирования[a]

Задача программирования	Код интелл./ канц.[b]	Оценка творчества
Общие задачи программирования		
Анализ требований	и	1,00
Анализ потоков данных	и	1,00
Функциональная декомпозиция	и	0,50
Разработка проектной документации и спецификаций	и	
Контроль обновления документов	к	
* Доступ к проектам многократного использования		
Разработка нового кода	и	
* Доступ к коду многократного использования		
Анализ и модификация имеющегося кода	и	0,25
Реструктуризация имеющегося кода	к	
Удаление мертвого кода	и	
Рецензирование проекта	и	0,25
Инспекция кода	и	
Персональная отладка	и	0,25
Создание контрольных примеров	и	0,50
Контроль библиотеки тестирования	к	
Анализ дефектов	и	
Создание документации для пользователей	и	
Создание электронной подсказки и учебников	и	
Обучение пользователей	и	

[a] Jones T. C. «Why Choose CASE?», *American Programmer*, 3, 1 January 1990, 14–21.

[b] и = интеллектуальная задача,
к = канцелярская задача,
н = неопределенная задача.

* Задачи, основанные на технологии, не рассматриваемые в данном анализе задач.

В табл. 3 столбец «Творческие задачи» содержит итоги по результатам, приведенным в табл. 1 и 2. В первом столбце показан диапазон числа задач, определенных как творческие четырьмя экспертами, от минимального количества задач, определенного кем-то из экспертов, до максимального. Следующие два столбца содержат количество задач, определенных как творческие хотя бы одним экспертом («минимальная поддержка») и двумя или более экспертами («существенная поддержка»). В последнем столбце представлено «взвешенное» среднее, полученное суммированием баллов творчества из табл. 1 и 2.

Таблица 3. Классификация задач CASE и не-CASE

	Интеллектуальные/ канцелярские задачи			**Творческие задачи**			
Класс задач	**Канцелярская**	**Неопределенная**	**Интеллектуальная**	**Диапазон**	**Миним. поддержка[a]**	**Существ. поддержка[b]**	**«Взвешенная» средняя[c]**
Задачи CASE табл. 1	14[d] (21)	11 (17)	40 (62)	0–18	21	6	7,25
Задачи не-CASE табл. 2 (а)	0	0	13 (100)	3–7	10	6	4,75
Табл. 2 (б)	3 (17)	0	15 (83)	2–5	7	4	3.75
Суммарно	17 (18)	11 (11)	68 (71)	6–28	38	16	15,75

[a] Количество задач, определенных как творческие хотя бы одним экспертом.

[b] Количество задач, определенных как творческие двумя или более экспертами.

[c] Сумма творческих оценок.

[d] В таблице приводится количество задач в каждой категории; в скобках указаны проценты.

Табл. 3 сообщает, например, что для задач CASE из табл. 1:

- Среди экспертов диапазон количества задач, оцененных как творческие, колебался от 0 (ни один эксперт не признал их творческими, поскольку они решались CASE-средствами) до 18 (из 40 интеллектуальных задач).

- Некоторую поддержку (хотя бы одного эксперта) как творческая получила 21 задача.
- Существенную поддержку в смысле творческого характера получили шесть задач.
- Взвешенное среднее количество творческих задач из табл. 1 составило 7,25.

Что можно увидеть в этих таблицах? Вероятно, главным наблюдением будет, что эксперты по-разному смотрят на задачи и их творческие аспекты. В противоположность согласованности, которой отличалась классификация задач на интеллектуальные и канцелярские, в данном случае согласия между четырьмя экспертами практически не было. Хотя и были случаи полного единодушия – все эксперты согласились, что «Анализ требований» и «Анализ потоков данных» в табл. 2 (б) являются творческими задачами, но оценки других задач существенно различались.

Например, некоторые задачи моделирования один эксперт считал творческими, а остальные – нет. Обратите внимание на преобладание в таблице оценок 0,25 – это случаи, когда лишь один эксперт счел задачу творческой («Миним. поддержка»).

Суммарные данные, представленные в табл. 3, также демонстрируют существенные разногласия. Проценты для творческих задач не приводятся, потому что числа по канцелярским, неопределенным и интеллектуальным задачам взяты из прежнего исследования и уже составляют в сумме 100%. (Обратите внимание, что задачи, охарактеризованные как творческие, почти всегда ранее были отнесены к категории интеллектуальных.) Однако в целом творческие задачи составляют примерно 16%, в диапазоне 6–29%. Широкий диапазон указывает, что у экспертов не было согласия по поводу того, какие задачи являются творческими.

Анализ результатов

Почему мнения экспертов столь сильно различаются? Это могло стать результатом действия нескольких факторов.

Прежде всего, различное толкование с точки зрения творчества допускают *определения задач*. Несмотря на их достаточную конкретность и расположение на достаточно низком уровне декомпозиции иерархии, они все же оставляют достаточное место для сомнений в том, какую работу они предполагают. Например, один эксперт многие задачи пометил знаком «?», не будучи уверенным в том, творческие они или нет, а другой во многих случаях поставил оценку «да/нет», подразумевая при этом, что они могут быть творческими или нет, в зависимости от того, как их интерпретировать. (Интересно отметить, что хотя эти «да/нет» были засчи-

таны в таблицах как «нет», отметивший данную неопределенность эксперт был как раз тем, кто больше всего задач оценил как творческие. Иными словами, если бы «да/нет» были засчитаны как «да», общий итог этого эксперта значительно превысил бы 29%.)

Толкования требуют не только определения задач, но и *определение творчества*. Несмотря на общепринятость нашего определения, в контексте некоторых задач эксперты сочли его применение затруднительным. Например: что представляет собой «новшество»? Или когда мышление оказывается «нетрадиционным»? Задача, которая одному кажется «неясной и некорректной», для другого может быть знакомой и привычной. Вероятно, для оценок такого рода существенное значение имеет личный опыт.

Наконец, *личные пристрастия* также могли повлиять на оценки экспертов. У большинства экспертов уже есть сложившееся мнение по поводу того, является ли та или иная работа творческой. (Попробуйте собрать жюри из коллег-профессионалов, которые беспристрастно относились бы к выбранной теме!) Поэтому при интерпретации отдельной задачи в условиях двусмысленного определения творчества эксперт мог склониться к имеющейся у него точке зрения, а не вырабатывать ее в ходе исследования.

Выводы

В статье представлены результаты исследования важной проблемы. Проблема получила некоторое освещение, но не была решена. Ясно, что какое-то количество задач программирования требует творчества. Столь же ясно, что согласия по поводу этого количества нет – по крайней мере, оно не обнаружилось в данном исследовании.

Несмотря на отсутствие окончательных результатов некоторые выводы, как нам кажется, можно сделать.

Во-первых, с практической точки зрения, тот факт, что творчество составляет существенную часть решения некоторых задач разработки программного обеспечения, должен опровергнуть взгляды вроде «я мог бы научить программировать даже свою матушку» и «можно разрабатывать программы без участия программистов», согласно которым работа программиста не требует больших умственных усилий. Эти исследования убедительно показывают, что при разработке программного обеспечения решаются, в основном, интеллектуальные задачи, часть которых (возможно, большая) требует творчества. Учитывая, что мы не умеем автоматизировать творчество и, скорее всего, никогда не сумеем, тот факт, что в этой работе необходима какая-то *доля* творчества, означает, что задачи программирования, вероятно, никогда не удастся полностью автоматизировать или поручить тем, кому не хватает знаний по программированию.

Во-вторых, с исследовательской точки зрения, тот факт, что творчество составляет важную часть работы по созданию программного обеспечения, должен побудить к дальнейшему изучению данной темы. Проблема определения того, в какой степени задачи программирования являются творческими, сохраняет свою важность, но пока не решена, а методы стимулирования творчества и управления им представляют большой интерес как для обучения программированию, так и для науки.

В третьих, с точки зрения финансирования исследований, то обстоятельство, что творчество играет большую роль в решении определенных задач программирования, должно поощрить организации, субсидирующие исследования, предоставлять поддержку тем направлениям, которые изучают автоматизацию автоматизируемых задач, а не автоматизацию любых задач программирования, что менее целесообразно, хотя и принято в настоящее время. Несомненно, попытки автоматизировать задачи, требующие творчества, имеют меньше шансов на успех, чем попытки автоматизировать решение более легких задач.

Полученные результаты явно подкрепляют высказанную в начале статьи точку зрения, согласно которой задачи программирования являются сложными (обратите внимание на одно лишь количество задач, перечисленных в таблицах, – целых 96!) и творчество – важный компонент программных решений и, скорее всего, им и останется. Свидетельства в пользу этой точки зрения не столь неоспоримы, как нам бы хотелось, но, тем не менее, они есть.

Благодарности

Авторы хотят поблагодарить Брюса И. Блюма (Bruce I. Blum), Денниса Ф. Галетту (Dennis F. Galletta) и Н. С. Юманата (N. S. Umanath) за помощь в выполнении этой работы.

Ссылки

COUGER 1990 – «Ensuring Creative Approaches in Information System Design», *Managerial and Decision Economics,* Vol. 11, pp. 281–295, 1990; J. Daniel Couger.

DEGRACE 1993 – *The Olduvai Imperative: CASE and the State of Software Engineering Practice,* Yourdon Press, 1993; P. DEGRACE and L. STAHL.

GLASS 1992 – «Software Tasks: Intellectual or Clerical?», *Information and Management,* Nov., 1992; Robert L. Glass, Iris Vessey, and Sue A. Conger.

MARTIN 1982 – *Application Development Without Programmers*, Prentice-Hall, 1982; J. Martin.

MBA 1990 – «More on MBAs in MIS», *Information Week*, Fax Forum, p. 56. March 19, 1990.

NEWELL 1962 – «The Processes of Creative Thinking», in *Contemporary Approaches to Creative Thinking*, H. Gruber, G. Terrell, and M. Wertheimer (eds.), Atherton Press, 1962; A. Newell, J. Shaw, and H. Simon.

USNRC 1990 – «Scaling Up: A Research Agenda for Software Engineering», Computer Science and Technology Board, U.S. National Research Council, 1990.

6.3. Почему люди «покупаются» на революционные идеи

Каждая новая идея в программировании, возникающая в мире науки или в мире поставщиков программного обеспечения, преподносится как решение всех проблем продуктивности и качества разработки программ.

И каждая такая новая идея на практике оборачивается посредственностью.

В чем тут дело? Может быть, практики-программисты слишком упрямы? Или глупы? Может, ученые и поставщики программного обеспечения наивны? Или ошибаются?

Какими бы ни были ответы на эти вопросы, сами они вполне обоснованны. На каждую из новых идей тратятся огромные средства. Но *где* же обещанные выгоды?

Как ни странно, ответы на эти вопросы есть, и они являются плодом серьезных размышлений. Конечно, есть и очень субъективные точки зрения. Исследователи и поставщики тайком и втихомолку (а иногда открыто и громогласно) соглашаются в том, что практики *действительно* невероятно упрямы и глупы. Весь так называемый «кризис программирования» держится на этом мнении.

Но в действительности такой взгляд совершенно не оправдан. Во-первых, нынешние практики – как и прежние – решили в этой новой области невероятно сложные задачи – отправив человека в космос, управляя экономикой во всем мире, совершенно изменив характер образования и индустрии развлечений и породив новые способы ведения боевых действий. Кроме того, сегодняшние практики – продукт вчерашней академической науки. (То есть большинство практиков прошли те самые учебные курсы, которые преподаются нашими исследователями.) Каким же козням практики удалось так быстро развратить наших студентов-практиков?

В то самое время, как исследователи и поставщики заламывают руки по поводу практиков, те не поднимают шума (изредка – выступают открыто и громогласно) и едины в мнении, что их противники чудовищно не осведомлены и ошибаются. «Оставьте нас в покое, – часто говорят они исследователям. – У нас и без *вас* достаточно хлопот».

Но такая позиция тоже не разумна. Во-первых, современные исследователи – как и прежние – очень способные люди, которые выдвинули идеи и предложили методы, в своем роде столь же впечатляющие, как приложения, созданные практиками. Если они не осведомлены и ошибаются на счет практики, значит, они просто не пытались разобраться в ней, потому что трудно представить, чтобы они пробовали это сделать и не смогли.

Почему же мы снова и снова встречаемся с этим явлением? Почему восторги по поводу новых технологий сменяются разочарованиями?

Думаю, выбрать отправную точку для поиска ответов нетрудно. Сначала нужно понять, в чем состоит истинная природа создания программного обеспечения, и это близко к заголовку данного раздела. Чтобы понять, почему прорывы в программировании столь часто оборачиваются провалами, нужно выяснить, в чем разница между интеллектуальной и канцелярской работой и как они соотносятся.

Прежде всего, рассмотрим такой вопрос: умеем ли мы сегодня писать программы, существенно усиливающие мыслительную способность человека?

Затем обратимся к другому вопросу: намного ли компьютеры облегчили человеку решение задач, требующих интенсивной работы мысли?

Полагаю, если ставить эти вопросы безотносительно проблемы создания программного обеспечения, ответы определенно будут «нет» и «нет». Компьютеры и программное обеспечение, несмотря на всю их магию, не являются инструментами, способными существенно поддержать человеческий разум в решении задач, требующих интеллектуальных усилий. Во всяком случае, в данный момент (эта точка зрения будет обоснована в главе 15 данной книги). Возможно, этого не случится никогда. Время покажет.

А как же все эти средства CASE и языки четвертого поколения, а также другие крупные технологические новшества в программировании, спросите вы? Мы *должны* рассматривать эти вопросы в контексте программирования. И тогда, как вы думаете, ответы могут быть иными.

Что ж, рассмотрим CASE-средства как особо интересный пример этого явления. В самый разгар ажиотажа по поводу CASE-революции появилась статья [Bird 1991], отчасти проливающая свет на данную проблему. Пусть эта статья была опубликована в популярной компьютерной прессе,

и автор не вполне понял важность сказанного им, тем не менее он проявил удивительную проницательность.

«Во многих компаниях, где применяются самые популярные CASE-инструменты, вся предварительная работа ведется на лекционной доске, – говорится в статье. – И лишь когда все правильно (или почти правильно) выглядит на доске, начинают применяться средства CASE». «То ли это, на что рассчитывало руководство, платя 10–15 тыс. долларов за программное обеспечение CASE? – спрашивает автор. – Руководство явно рассчитывало на более высокую эффективность этих инструментов в поддержке творческих этапов разработки».

«Творческих этапов» – в этом вся суть. Программное обеспечение покупалось, как полагает автор статьи, для поддержки мыслительных процессов человека. И он, конечно, прав. Руководитель, потративший крупную сумму, рассчитывал – вероятно, полагаясь на обещания продавца, в свою очередь основанные на заверениях исследователей, – что эти инструменты облегчат основную интеллектуальную или даже творческую работу при разработке программного обеспечения.

А они этого не делают. В этом и проблема, и истина. Большинство CASE-средств – особенно верхнего уровня, предназначенные для помощи в системном анализе и проектировании (несомненно, наиболее интеллектуальных и творческих этапах жизненного цикла) – не могут выполнить вместо человека требуемую интеллектуальную работу и никогда для этого не предназначались. Они помогают системным аналитикам и проектировщикам программного обеспечения создавать, модифицировать и хранить *представления* результатов системного анализа и проектирования.

Повторим еще раз, поскольку это важно. Большинство CASE-инструментов верхнего уровня предназначено для поддержки процесса представления, а не интеллектуального процесса создания представлений. Это два совершенно разных вида работы. В одном случае нужно понять проблему и предложить способ ее решения, и это глубоко интеллектуальный процесс. В другом случае делаются чертежи, составляются спецификации и проект, и это относительно канцелярская работа.

С учетом сказанного, становится понятным ряд вещей. Во-первых, руководители, купившие CASE-инструменты, просто исходили из неверных предположений. Если они предполагали, что покупают средства, усиливающие умственные способности, то ошибались.

Во-вторых, разработчики, чертившие на доске, прежде чем обратиться к CASE-инструментам, вполне могли быть правы. Они усиливали свои интеллектуальные процессы с помощью средства, которое: а) способствовало совместной работе и б) ускоряло пересмотр и переработку. Большинство современных CASE-средств плоховато справляются с обеими задача-

ми. При этом многие согласится, что групповые и эвристические процессы составляют сердцевину этапов разработки программного обеспечения, требующих напряженных интеллектуальных усилий.

Что из всего этого следует? Мы в своей области – программировании – плохо разобрались с тем, какая часть нашей работы требует умственных усилий, а какая является преимущественно канцелярской. В возникшей из-за этого путанице мы ошибочно принимаем средства, поддерживающие один вид работы, за средства, способные помочь нам в другом виде.

Еще хуже, что мы повторяем ту же ошибку снова и снова. Мы рассчитываем, что языки четвертого поколения (4GL) помогут нам справиться с нехваткой программистов, и удивляемся, почему эта нехватка еще острее по сравнению с временами, предшествовавшими появлению 4GL. Мы рассчитываем, что CASE-средства позволят автоматизировать программирование, особенно в части системного анализа и проектирования, а потом удивляемся, почему они в значительной своей части оканчивают свою жизнь на полке. Мы рассчитываем, что объектно-ориентированное программирование станет для нас очередной панацеей, сведя программирование к простым задачам, и с растущим удивлением узнаем, что для объектно-ориентированной разработки требуются, пожалуй, даже более подготовленные и способные программисты, чем прежде.

Мы тратим огромные средства на революционные идеи, которые в итоге приносят скромные улучшения – считается, что около 5–30%, – вместо существенного роста производительности, на который мы рассчитывали. В результате мы становимся недовольными, разочарованными и подозрительными. Высшее руководство, не дождавшись обещанного ему освобождения от программистов, проявляет беспокойство, потом сердится, потом теряет всякое доверие к компьютерным организациям.

Возникает порочный круг прорывов и провалов, из которого не видно выхода. Думаю, причина в том, что мы достаточно плохо понимаем свою сферу деятельности и не умеем выделить среди всех задач программирования те, которые требуют сложной умственной работы. Пока практикующие программисты не разберутся в том, что они покупают, пока поставщики не станут честно и точно описывать то, что они продают, и пока исследователи не начнут тщательно проверять выдвигаемые ими идеи, мы не выйдем из этого круга, в котором за обещанным прорывом следует очередной провал. И мы будем мешать тому постепенному прогрессу, который действительно возможен в нашей области.

Ссылки

Bird 1991 – «CASE Crop a Flop», *Software Magazine,* Nov. 8, 1991; Chris Bird.

Глава 7 | Теория и практика

Введение
7.1. Временное соотношение теории и практики
7.2. Еще о соотношении теории и практики
7.3. Теория и практика: тревожный пример
7.4. Полет шмеля
7.5. Теория и практика: разные жалобы
7.6. Области программирования, где практика опережает теорию

Введение

Никто не знает, как арабский архитектор проектировал [купол Соборной мечети в Исфахане], но он сумел создать купол почти идеальных пропорций за несколько столетий до того, как появились математические средства, необходимые для решения этой задачи.

Гораций Фриленд Джадсон

«The Search for Solutions»,
Holt, Rinehart and Winston, 1980, приводится
с разрешения Henry Holt and Company, Inc.

Греки возносили науку выше вульгарной практики, доходя до педантизма, допускающего интеллектуальную распущенность, что иногда имело прискорбные последствия.

Ной Кеннеди

«The Industrialization of Intelligence»,
Unwin Hyman, 1989

Выбирая одну из сторон в споре о способах разработки программного обеспечения, оказываешься в довольно неожиданной компании.

Если считать, что доминируют два подхода:

- дисциплина/формализм/оптимизация/количество/процесс/канцелярия и
- гибкость/эвристики/разумная достаточность/качество/продукт/интеллект

(а неплохо здесь смотрятся названия всех глав первой части этой книги!), то обнаружится, что большинство теоретиков выбрали для себя первый список, а практики – опять-таки большинство – второй.

Но не кажется ли это несколько странным? Ведь как работают теоретики: в своей работе они применяют все то, что *осуждают* у практиков, избегая всего того, что сами им *рекомендуют*.

Попробуйте, например, представить дисциплинированного формалистичного теоретика, который рассчитывает получить в своем исследовании оптимальные количественные результаты с помощью преимущественно канцелярских методов. Это просто невозможно.

Так почему же теоретики навязывают практикам программирования полную противоположность собственного стиля? Многие годы я размышлял об этом, но так и не нашел приемлемого ответа.

Может быть, они искренне считают программирование настолько простым делом, что все эти дисциплинированные, формальные и даже канцелярские методы должны работать?

Может быть, сами не желая применять эти методы, они на самом деле не видят слабости дисциплинированных, формальных, канцелярских подходов?

Может быть, они не хотят поверить в то, что – как пытались убедить нас Брукс и Парнас – программирование является самым сложным видом человеческой деятельности, потому что это принизило бы то, чем занимаются они сами?

А может быть, поиск формальных дисциплинированных методов – настолько увлекательная задача, что они и не задумываются о том, в какой мере это применимо к практике программирования?

Я больше склоняюсь к последнему ответу. В конечном счете теоретик больше всего стремится найти интересную (а вовсе не полезную) работу (эта тема возникнет еще раз в одном из разделов другой главы). Что может быть увлекательнее изобретения языков и формализмов или придумывания дисциплины и методологий, с которыми должен работать кто-то другой? А если и возникнет слабое подозрение, что эти методы, несмотря на всю их привлекательность, не нужны в практической деятельности, его легко погасить, читая труды других теоретиков – а их предостаточно, – в которых практика принижена и выглядит простой и автоматизируемой. В конце концов, большинство теоретиков никогда не занималось практикой и редко имеет право выносить суждения по ее поводу.

Однако придется признать, что по мере движения в направлении методов создания проблемно-ориентированного программного обеспечения и роста умения решать очень узкие задачи в очень узких областях может случиться так, что в каких-то областях найдутся такие проблемы, которые можно решать с помощью строгих, формальных и даже канцелярских методов. В конце концов, что представляют собой языки 4GL, как не проблемно-ориентированные языки генерации отчетов в прикладной области доступа к базам данных? И разве не подходят в данный момент языки 4GL ближе всего к тому, чтобы свести программирование к автоматизируемой

задаче? Поэтому набору принципов, поддерживаемых теоретиками, есть некоторое оправдание и со временем оно вполне может усилиться.

Но это дело будущего, а в настоящее время состояние проблем – можно сказать, «теоретических представлений о практике», – по которым проходит водораздел, полагаю, таково:

- Дисциплина и гибкость. Дисциплина важна при работе над крупными программными проектами и полезна при работе над средними, но для малых она может оказаться непосильным грузом. Гибкость? Она важна для *каждого* из этих классов проектов. «Странная парочка» будет жить!
- Формальные методы и эвристики. Формальные методы просто никак не показали себя в больших или хотя бы средних проектах, и хотя их можно применять в малых проектах, свидетельств их особой полезности практически нет. (И это после 35 лет исследований в этой области!) Напротив, эвристические методы применимы в проектах любого масштаба.
- Оптимизация и разумная достаточность. Оптимальные решения, как бы старательно их ни искали, обычно отсутствуют в случае сложных и важных задач. На практике довольствуются разумной достаточностью. (Конечно, все зависит от решаемой задачи. Некоторые задачи, например, в области информационных систем, либо решаются оптимально, либо не решаются вовсе. Но, скажем, для предсказания погоды это не так.)
- Количество и качество. На практике поиску количественного решения отдается предпочтение – даже когда это ошибка! Разумеется, поиск верных количественных ответов часто оказывается неудачным, потому что есть много вещей, которые нельзя осмысленно измерить. Странным и опасным образом качественные ответы иногда маскируются под количественные и используются ошибочно.
- Процесс и продукт. В практике программирования всегда уделялось внимание процессу, начиная с тех времен, когда его называли «технологией» или «методами». (Например, в первых номерах «Communications of the ACM» была колонка «Pracniques».) Работа Software Engineering Institute по моделированию процесса привлекла в последние годы большее внимание к этой задаче и в значительной мере ориентировала ее в сторону управления. Однако центром внимания практики был, есть и будет продукт.
- Интеллектуальные и канцелярские задачи. На практике преобладает интеллектуальная деятельность, находясь, как было показано в предыдущей главе, на уровне примерно 80%. Есть и канцелярские задачи, например управление конфигурациями, регистрация модифика-

ций, выполнение контрольных примеров и пр.; они имеют очень большое значение, но их объем гораздо меньше.

Такова моя личная теория практики программирования. Интересно было бы сравнить ее с вашими теориями.

В разделах этой главы проблема соотношения теории и практики будет исследована подробнее, но под другим углом. Я хочу показать, в каком состоянии находятся теория и практика и как они связаны друг с другом. Думаю, кое-что здесь вас удивит.

7.1. Временное соотношение теории и практики

Теория: изложение основополагающих принципов предмета.

Практика: действие как противоположность теории; активное занятие профессиональной деятельностью.

Словарь Oxford American Dictionary, 1980

Смысл слов «теория» и «практика» достаточно понятен и общепринят, поэтому мы редко сомневаемся в том, что имеет в виду тот, кто их употребляет. Но что можно сказать о временном соотношении между этими двумя понятиями: что появилось раньше – теория или практика?

Большинство из нас, проведя в школе десять или более лет, даст ответ автоматически: теория предшествует практике и определяет ее. Но этот автоматический ответ может оказаться весьма далеким от истины, свидетельствуя о глубоких заблуждениях.

Возьмем, например, следующую цитату из «Notes on the Synthesis of Form» (Заметки о синтезе формы) Кристофера Александера (Christopher Alexander), Harvard University Press, 1964:

> Особый профиль крыла, благодаря которому могут летать самолеты, был изобретен как раз тогда, когда удалось «доказать», что аппараты тяжелее воздуха летать не могут. Его аэродинамические свойства стали понятны лишь позднее, когда он какое-то время уже применялся на практике. Фактически, изобретение и использование профиля крыла внесло существенный вклад в развитие аэродинамики, а не наоборот.

Таким образом, согласно Александеру в данном случае практика предшествовала теории. Нет ли других примеров таких неожиданных ситуаций?

Вот еще одна цитата, поразительно напоминающая случай, приведенный Александером, взятая из текста, принадлежащего Д. Д. Прайсу (D. D. Price) и опубликованного в Записках ежегодной конференции американской ассоциации содействия развитию науки под названием «Sealing Wax and String: A Philosophy of the Experimenter's Craft and Its Role in the Genesis of High Technology» (Сургуч и веревочка: философия мастерства экспериментатора и его роль в возникновении высоких технологий), Proceedings of the American Association for the Advancement of Science Annual Meeting, 1983:

> Термодинамика обязана паровому двигателю в значительно большей мере, чем паровой двигатель – термодинамике... Если взглянуть на обычный исторический ход событий,... то случаи, когда технология оказывается прикладной наукой, очень немногочисленны. Гораздо чаще наука оказывается прикладной технологией.

Два примера того, как практика предшествует теории, не могут, конечно, служить доказательством или хотя бы указывать на тенденцию; однако Прайс считает, что тенденция, возможно, есть. Нужны еще примеры?

Герберт Саймон (Herbert A. Simon) «The Sciences of the Artificial» (Науки об искусственном), The MIT Press, 2nd Edition, 1981:

> ...Главный путь к разработке и усовершенствованию систем с разделением времени состоял в том, чтобы построить их и посмотреть, как они себя поведут. Именно так и поступили. Последовательно строили, модифицировали и усовершенствовали системы. В принципе, теория могла бы предсказать результаты этих экспериментов, сделав их излишними. Но она этого не смогла, и я не знаю никого достаточно близко знакомого с этими чрезвычайно сложными системами, кто бы конкретно предложил, как это сделать. Чтобы понять эти системы, нужно было их построить и посмотреть, как они себя ведут.

После этой цитаты из Саймона идея о первичности практики, а не теории подвигается ближе к нашим родным местам – компьютерам и программам. Посмотрим, что еще нам известно о теории и практике в программировании.

Компьютерная и программная практика росла у меня на глазах, и я помню многие определяющие события в этой сфере. Мне близка идея практики, предшествующей теории. Появление компьютеров связано, конечно,

с первыми исследованиями, проводившимися в многочисленных лабораториях Северной Америки и Европы. Однако в середине 1950-х началось бурное развитие компьютерной техники и программирования как профессиональных областей. Если вначале практика и не опережала теорию, то с течением времени, несомненно, ее обошла. Компьютерные науки как учебный предмет появились лишь в 1960-е. А теории, ставшие плодом этих научных усилий, появились, пожалуй, не раньше конца 1960-х – начала 1970-х.

Таким образом, по личным наблюдениям ветерана этой области, идея о том, что практика предшествует теории, явно выглядит правдоподобной. Но, разумеется, личный опыт бывает обманчив. У кого-то может быть совершенно иной личный опыт. Нет ли других способов изучить этот вопрос?

До настоящего времени я говорил о компьютерах и программировании как о единой дисциплине, которую можно изучать в целом. Но что если рассмотреть отдельные составляющие элементы этого целого? Не поможет ли это выявить какую-нибудь систему в отношениях между практикой и теорией?

Возьмем, к примеру, стили программирования. Что здесь появилось раньше – практика или теория? Конечно, множество программ появились прежде книг о стиле программирования. Прекрасно помню, что у одних программ был великолепный стиль, а у других – нет. Полагаю, стиль был достаточно хорошо разработан на практике, прежде чем появилась теория стиля. (Интересно отметить, что одна из первых серий книг, посвященных стилю, «Proverbs...» Ледгара, по существу систематизировала положительный практический опыт.)

А как насчет разработки компиляторов? Практика породила компиляторы таких языков, как Фортран, Commercial Translator, FACT и, позднее, Кобол, намного раньше, чем появилась вразумительная теория написания компиляторов. И здесь тот же случай, полагаю: на практике написание компиляторов – один из главных предметов нынешней компьютерной науки – было развито достаточно хорошо еще до появления теории.

Даже сегодня, бывает, практика опережает теорию. На системном уровне часто применяется моделирование для уточнения требований и проектирования при решении сложных задач. Тем не менее, оно редко становится предметом научных исследований. (Заметим, что в том виде, как оно используется на практике, моделирование нацелено на создание пробного практического решения с целью разработки теории проблемы. Даже здесь практика опережает теорию!)

Проектирование интерфейса пользователя, хотя и уходит корнями глубоко в теорию, разработанную Xerox PARC, сейчас быстрее развивается благодаря компьютерной практике, чем поддерживающей его теории.

Теория для собственно проектирования все еще плохо проработана. Курсы по проектированию часто ориентированы на методологию и представление, тогда как большинству проектировщиков известно, что это гораздо более сложная область, отнюдь не ограничивающаяся общей схемой и средствами фиксации. И здесь практика значительно опережает теорию.

Также, фактически, общее понятие о решении задач – чем на самом деле и занимается программный инжиниринг, – все еще находится на ранних этапах теоретической разработки (см., к примеру, «Науки об искусственном» Саймона, цитировавшиеся выше), хотя на практике люди занимаются решением задач (в том числе проектированием) не одну сотню лет.

То есть примеров практики, предшествующей теории, довольно много. Вы удивлены? Я и сам не перестаю этому удивляться, несмотря на все исторические факты. Меня всегда учили, что теория создает основу, благодаря которой может успешно действовать практика. Если эта идея ошибочна, то, пожалуй, надо рассмотреть последствия.

Изобразим связь между теорией и практикой, которую подразумевает эта идея. Я попытался это сделать на приведенном здесь рисунке. Суть в том, что по моим представлениям для каждой отдельной дисциплины впереди идет практика, которая вначале довольно быстро развивается; теория возникает, когда практика дает достаточно материала для формализации, и тоже развивается быстро; с какого-то момента теория начинает опережать практику (рис. 7.1.)

Предположим, что эти отношения изображены верно. Каковы будут последствия? Во-первых, из этого графика следует, что на ранних этапах развития дисциплины теория сможет развиваться успешнее всего, если станет изучать практику. Истина в том, что старые методы не должны мешать теоретикам формулировать новые идеи, и все же многое можно узнать, изучая практическую деятельность, особенно на лучших образцах. Это важная мысль. Как правило, развитие теории компьютеров и программирования не сообразовывалось с ее последствиями. Редкие теоретики компьютерных наук были в прошлом практиками. В нашей сфере редки экспериментальные, моделирующие практику исследования. И, не считая нескольких эмпирических исследований, развитие практиками теории изучается очень слабо.

Из этих размышлений следует, что подход к развитию теории нужно изменить, по крайней мере, на ранних этапах. Слишком богат запас знаний, накопленный практикой, чтобы теория могла его игнорировать.

Из этого графика можно также сделать вывод, что в какой-то момент практика должна начать прислушиваться к теории, которая ее обогнала. Например, теоретические достижения в области баз данных и структур данных значительно превосходят уровень знаний большинства практи-

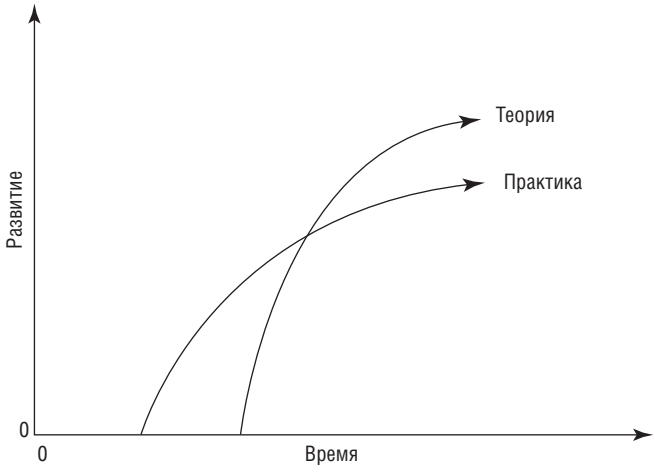

Рис. 7.1. *Теория плюс практика*

ков в этих областях. Практика здесь не на высоте. Как теория не умеет изучать практику, когда это необходимо, так и практика не желает прислушаться к теории в нужный момент.

Иными словами, существуют фундаментальные проблемы взаимодействия теории и практики, что и показал приведенный график, и от понимания его смысла напрямую зависит состояние мастерства и практики применения компьютеров и программирования.

Возможно, этот график слишком упрощен. В более точном представлении развитие практики и теории выглядит как чередование этапов, на каждом из которых верх берет то одна, то другая сторона. Но даже на таких более сложных картинках можно будет найти участки (например, с помощью увеличительного стекла), для которых справедлив более простой график.

Значит, практика и правда предшествует теории? На каком-то уровне и в какие-то моменты времени – да. Пора и практике, и теории извлечь выводы из этого обстоятельства.

Благодарности

Хочу поблагодарить Айрис Весси (Iris Vessey) и Дэйла Доузинга (Dale Dowsing) за помощь в развитии этих идей.

7.2. Теория против практики – новый взгляд

Надо полагать, у каждого, кто прошел нашу систему обучения и занялся профессиональной деятельностью, возникает свое небезынтересное мнение по поводу того, чему его учили и чему следовало бы учить.

Лично для меня путь от обучения к практике шел через чистую математику. Начав работать в сфере программирования, я с ужасом обнаружил, что все, чему меня учили в высшей школе, не имело никакого отношения к тем заданиям, которые я получал на работе.

С тех пор на протяжении всей своей карьеры я интересуюсь взаимоотношениями теории и практики. Я вижу, что между ними существует огромный «дефицит общения» и, как следствие, глубокое взаимное непонимание. И я вижу, что как теоретические, так и практические результаты жестоко страдают от такого недостаточного общения и взаимонепонимания. Этот разлад отрицательно сказался и на результатах моей собственной работы.

Мои переживания по этому поводу вылились в статью «Временное соотношение теории и практики» (см. предыдущий раздел этой главы). В ней я представил взаимоотношения между теорией и практикой так, как показано на рис. 7.1: сначала практика опережает теорию, а потом теория догоняет и перегоняет практику.

Впоследствии я продолжал думать над тем, что описал в этой статье. Думаю, основная ее мысль – что практика нередко предшествует теории и помогает ее возникновению – по-прежнему верна. Но, как часто бывает в жизни, отношения оказываются более сложными, чем это показано на рис. 7.1.

Немного о событиях, заставивших меня снова задуматься над графиком, приведенным на рис. 7.1. Это случилось на Всемирном конгрессе по вычислительной технике в Сан-Франциско, где я оказался в компании великих. Два докладчика обратились к этой теме – два блестящих мыслителя и оратора, каких я прежде не слыхал. Оба они, хотя и разными словами, изложили по существу одну и ту же мысль.

Дональд Кнут (Donald Knuth) из Стэнфордского университета, один из главных докладчиков, заявил: «Теория и практика могут и должны присутствовать одновременно». И далее сказал, что стремление слить их воедино «было главным кредо моей жизни».

Хайнц Земанек (Heinz Zemanek), бывший глава Венской лаборатории IBM, выступивший с докладом «Formal Structures in an Informal World» (Формальные структуры в неформальном мире), сказал: «Формализовать можно только то, что понято на элементарном уровне. Нужно начинать неформально, а затем двигаться к формализму».

Иными словами, неформальное практическое мышление должно предшествовать формулированию теории – и развиваться вместе с ней.

Когда Кнут привел примеры того, как он действовал в соответствии со своим кредо, я вспомнил важный случай из собственной жизни. В середине 1960-х, начав писать программы для научных исследований, я впервые получил задачу создать приложение для работы с данными – требовалось написать три разных генератора отчетов. Было бы неплохо отыскать в них сходство, и когда я бился над подведением итогов и формированием отчетов – все, кто занимался обработкой данных, сталкивались с этими задачами и уже научились их решать, – то заметил, что в организации этих трех продуктов есть какое-то общее зерно. В результате, завершив работу над выпуском отчетов, я получил один из первых пакетов – генераторов отчетов. В нашем контексте важно отметить, что моя работа над конкретными генераторами отчетов явилась важной прелюдией к тому, чтобы я смог разглядеть за ними более общую задачу. Иными словами, без неформального практического решения этих трех первых задач было бы невозможно выработать формальное, теоретически обобщающее решение.

Значит, предшествующие заголовок и рисунок нуждаются в важных поправках. Думаю, с учетом слов Кнута и Земанека в заголовке вместо «Теория против практики» должно быть «Теория плюс практика», а рис. 7.1 лучше заменить на рис. 7.2.

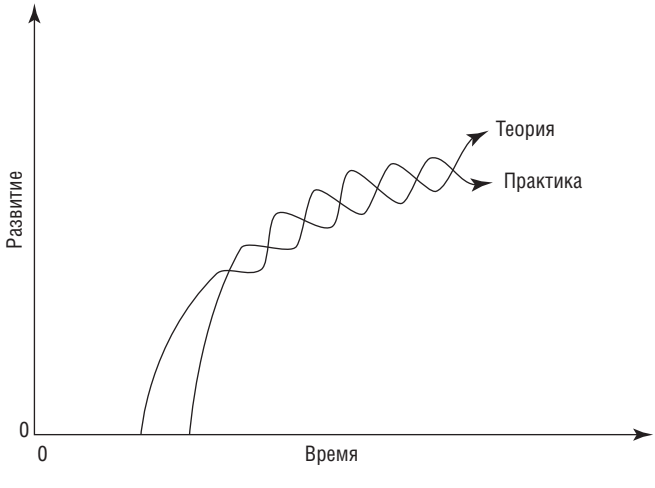

Рис. 7.2. *Теория плюс практика*

Теперь видно, что хотя первый проблеск новой идеи рождается в мире практики (не зря говорят «нужда научит»), взрослеть теория с практикой должны вместе. В противном случае, как это происходит в области компьютерной науки и программирования, неизбежны ослабление теории и застой в практике.

В заключение Кнут предложил аудитории очаровательную «нестандартную задачу», призванную побудить каждого к изучению собственного сплава теории и практики: выясните, чем занимается ваш компьютер в течение произвольно выбранной секунды, и проанализируйте результат. Предлагалось обдумать следующие вопросы:

1. Корректны ли программы, которые выполнялись? (Кнут сказал, что в большинстве из тех, что он видел, есть хотя бы одна ошибка.)
2. Применяются ли в этих программах существующие теории?
3. Могут ли существующие теории помочь этим программам?
4. Смогла бы помочь этим программам какая-нибудь новая теория?

Земанек тоже высказал напоследок замечательную мысль. «Пятьдесят лет назад, – сказал он, – были те, кто мастерски владел языком, то есть умел ясно выражаться и даже вдохновлял на это других. Сейчас таких людей нет, и это очень опасно». Причина, по его словам, в том, что система образования слишком увлеклась формальными методами и пренебрегает такими неформальностями, как здравый смысл и общение на естественном языке.

Может быть, Кнут и Земанек, а также все, кого это волнует, помогут развиться здоровому рабочему сотрудничеству теории и практики.

7.3. Теория и практика: тревожный пример

Философский вопрос о взаимосвязи теории и практики обсуждался в предыдущих главах этой книги. Здесь я хочу проиллюстрировать эту взаимосвязь примером из реальной жизни.

Несколько лет назад на престижной компьютерной конференции был представлен доклад на весьма практическую тему: как решить, когда следует прекратить сопровождать старую программу и переписать ее заново. Это один из главных вопросов всей проблемы сопровождения – с ним сталкиваются все разработчики и так или иначе решают его на протяжении более 30 лет. В некоторых компаниях, например Ford Motor Company и Pacific Telephone, для этих целей разработали сложный механизм принятия решений.

Но тот доклад был написан теоретиками. В его основу было положено исследование, проведенное на факультете информационных технологий

крупного университета и нимало не учитывающее 40 лет практической работы, связанной с принятием решений о том, чтобы «переписать или сопровождать». Исследователи написали свой доклад, изучив опубликованные описания программных проектов.

Каков был результат? Некоторые ученые, выслушав доклад, поздравили его авторов с хорошо выполненной работой. В конце концов, взявшись за важную тему, те проявили мудрость и понимание.

Но другая часть аудитории, включая меня (и, к моей радости, официально назначенного оппонента доклада), ужаснулась. Доклад не только не учитывал накопленный в отрасли опыт, но и содержал явно ошибочные выводы.

Кстати, стоит привести здесь выводы. Исследователи построили свой анализ решения о переписывании заново или продолжении сопровождения исключительно на математической оценке суммарной стоимости сопровождения программного продукта. Они предположили, что в тот момент, когда продукт требуется переписать заново, стоимость его сопровождения должна резко возрасти. Но это представление наивно. Часто необходимость переписать продукт возникает потому, что рамки первоначального проекта оказываются слишком тесными, делая дополнительное усовершенствование просто невозможным. И в этот момент стоимость сопровождения не только не возрастает, но фактически падает!

Исследователи сделали и другие, довольно любопытные выводы:

1. Программный продукт должен быть переписан заново, «не доходя до середины запланированного для него срока жизни».
2. Срок эксплуатации программного продукта:
   a) сокращается с ростом его начального размера (чем больше программа, тем короче ее жизнь);
   b) сокращается с увеличением структурированности (чем больше программа структурирована, тем короче ее жизнь);
   c) увеличивается с каждым новым переписыванием (чем чаще программу переписывают, тем дольше она живет).

Некоторые из этих выводов совершенно противоречат интуиции и практике. Неужели исследователи, получив такие результаты, не усомнились в корректности проведенного ими анализа?

В чем тут дело? Есть вопросы, в которых практика значительно опережает теорию. В этом тысячелетии (как и в прошлом) одним из таких вопросов остается сопровождение программного обеспечения. Исследователи должны знать, какие это вопросы, и при построении теорий опираться на

практику. Большинство исследователей не только не делает этого, но, похоже, не подозревает, что это можно или нужно делать.

В результате возникают слабые теории, которые практика со смехом отвергает. И наоборот – в практике случаются застои, когда не используются законные выводы науки. Ни теория, ни практика, от таких ситуаций не выигрывают.

Какова была моя личная реакция? Я спокойно сказал исследователю, что ему не следует продолжать работу в этом направлении, пока он не изучит накопленный практический опыт. И упомянул пару практиков программирования, которые могли бы ему помочь.

Мне было интересно, последует ли он моему совету. Чтобы выйти на контакт с теми, кого я назвал, ему пришлось бы связаться со мной, воспользовавшись визитной карточкой, которую я ему оставил.

На связь со мной он так и не вышел.

7.4. Полет шмеля

Впервые я услышал это в четвертом классе и с тех пор слышал неоднократно.

Согласно этому изречению, «по данным науки шмель летать не может».

Мы понимаем это так: наука не в силах объяснить, почему толстая, тяжелая пчела с крыльями-обрубками может летать.

Как бы вы ни воспринимали это странное заявление, ясно одно. Вряд ли кто-то, обдумывая загадочный полет шмеля, решит, что ошибается практика, то есть полет. Интуиция говорит нам, что ошибка, скорее всего, вкралась в теорию.

Этот пример, конечно, следует понимать в широком смысле.

7.5. Теория и практика: разные жалобы

Взаимосвязь теории и практики – сложная проблема. Но кроме сложности интерес здесь вызывает количество различных точек зрения – и жалобы их приверженцев.

Вот слова покойного Гарлана Миллза (Harlan Mills) из очаровательного интервью с ним, опубликованного в IEEE Software в июле 1988 года: «Очень жаль, что аппаратная часть развивалась так быстро. Понимаете, поработав с этими новыми процессорами... хотя бы лет пятьдесят, люди действительно смогли бы научиться писать логичные программы. На практике же, к тому времени, когда Дейкстра предложил структурное

программирование, оказалось, что все кому не лень работают на микропроцессорах с обработкой прерываний, и никакой теории для этого нет, но всякие там IBM, DEC, CDC и прочие – все они рвутся вперед, производя то, с чем еще никто не умеет работать».

Потрясающе – признавать то, что на ранних этапах практика опережает теорию, и одновременно сетовать на это! Как есть практики, остановившиеся в развитии и недовольные тем, что теория им мешает, так есть и теоретики (свидетельство чему – слова Миллза), которые говорят: мир, остановись, ты движешься слишком быстро, чтобы наши теории могли за тобой угнаться! Конечно, обе позиции ошибочны. Мир развивается с той скоростью, с какой может, подстегиваемый попеременно то практикой, то теорией, и никто не в силах остановить его. Да и стоило бы делать это?

Разумеется, не все теоретики придерживаются того же взгляда, что Миллз. Совершенно противоположное недовольство выказал Манфред Кохен (Manfred Kochen) [Kochen 1985] в статье, озаглавленной «Are MIS Frameworks Premature?» (Не преждевременны ли каркасы информационных систем?). В ней Кохен говорит, что исследователи пошли в ошибочном направлении, предположив, что построение системы определений должно предшествовать теории и экспериментам, а приложение теории должно происходить в последнюю очередь. Он приводит пример математического понятия производной, которая сначала использовалась, затем была открыта, потом исследована и развита и наконец определена, как правильную модель развития теории в любой области. Фактически жалоба Кохена противоположна миллзовской: поиск системы определений, с учетом состояния практики, является преждевременным, и «такие поиски ослабляют усилия, направленные на решение задач, которые, несмотря на несовершенство определений и отсутствие системы, скорее могут принести плоды».

Есть и жалобы другого рода. Дэвид Парнас в неожиданной статье, посвященной компьютерному образованию [Parnas 1990], обнаружил крупные недостатки системы образования, многие из которых связаны все с той же проблемой отношений между теорией и практикой. «В университетах и на крупных конференциях, – сказал он, – часто разворачиваются сражения между «теоретиками» и «прикладниками». Эти битвы, – многозначительно заключает он, – свидетельствуют о том, что у нас есть проблемы».

Какие проблемы видит Парнас? Самые различные. «Компьютерные науки, – говорит он, – слишком сосредоточились на узких научных интересах своих отцов-основателей... Разрыв между академическими компьютерными науками и реальным применением компьютеров стал слишком велик. Учебные программы не обеспечивают выпускников фундаментальными знаниями, необходимыми для долгосрочного профессионального роста».

Обращаясь далее к значению практики для образования и теории, Парнас говорит: «Студенты лишены той обратной связи, которую дает правильное или неправильное использование продукта, его отклонение или модификация. Это отсутствие обратной связи очень плохо сказывается на образовании».

В другой части компьютерного леса мы слышим нечто подобное от Питера Кина (Peter G. W. Keen) [Keen 1980], специалиста по информационным системам. «Исследованиями в области MIS должен заниматься только тот, – прямо заявляет он, – кто в совершенстве овладел какой-нибудь компьютерной технологией или методологией... Стремление стать «ученым» может быть похвально, но если оно явно сочетается с презрением к «применению», «бизнесу» или «практике», возникает сомнение в этичности пребывания такого исследователя в бизнес-школе».

В заключение своей статьи Кин недвусмысленно заявляет, что «...мир практики находится в центре, а не на периферии» теории и исследований MIS.

В еще более эмоциональной и полемичной статье Растом Рой (Rustom Roy), слова которого приводит «Newsweek» [Roy 1991], оспаривает саму ценность научных исследований: «Многие программы вывернуты наизнанку. Они начинают с фундаментальных принципов и лишь в конце подходят к приложениям». Рой возражает против такого подхода: «Только то, что связано с жизнью, можно запомнить на всю жизнь».

Возражения Роя иногда кажутся не просто жалобами. Он критикует качество *существующих* теоретических разработок, отмечая, что из публикуемых в различных научных областях статей, имеющих высокий (по мнению рецензентов) уровень, 45% ни разу не цитируется другими исследователями в течение пяти лет, следующих за публикацией. «Смысл в том, – говорится в «Newsweek», – что почти половина научной работы, проводимой в нашей стране, не имеет практически никакой пользы». Рой называет исследователей, претендующих на правительственное финансирование, «королевами на пособии»[1] в белых халатах».

Не все жалобы столь эмоциональны. Однако из сказанного ясно, что соотношение теории и практики – весьма спорный вопрос, по которому есть резкие мнения.

---

[1] Welfare Queens («королевы на пособии») – женщины, не имеющие мужа, работы и профессии, с несколькими детьми, часто от разных отцов; одна из целевых групп избирателей в предвыборных кампаниях в США 1990-х. – *Прим. перев.*

Постскриптум

Приведу имеющую отдаленное отношение к нашему обсуждению цитату из Стивена Джонса (Steven Jones), одного из физиков Brigham Young University, участника сомнительных экспериментов по ядерному синтезу: «У нас есть эксперимент, но нет теории. У нас есть Золушка, но нет ее туфельки». Можно ли назвать практику Золушкой без туфельки? Или это больше относится к науке?

Ссылки

KEEN 1980 – «MIS Research: Reference Disciplines and a Cumulative Tradition», *Proceedings of the International Conference on Information Systems,* 1980; Peter G.W. Keen.

KOCHEN 1985 – «Are MIS Frameworks Premature?», *Journal of Management Information Systems* Vol. II, No. 3, 1985; Manfred Kochen.

PARNAS 1990 – «Education for Computing Professionals», *IEEE Computer,* January 1990; David Lorge Parnas.

ROY 1991 – «Gridlock in the Labs», *Newsweek,* January 14, 1991; from an interview with Rustum Roy.

7.6. Области программирования, где практика опережает теорию

Уже несколько лет я повторяю, что в новых областях практика иногда опережает теорию. Я заявлял это в печати и лично перед многочисленными аудиториями. Все, перед кем я выступаю, – как практики, так и теоретики, – обычно слушают меня с глубоким интересом, но иногда с несколько скептическим выражением – дескать, все это очень интересно, но стоит ли верить этой чепухе?

Но однажды во время моего выступления некий особенно заинтересованный и хмурый слушатель задал мне непростой вопрос. «Ладно, – сказал он, – это все интересно, но будьте конкретны. *Где* именно в вычислениях и программировании практика опережает теорию?»

Законный вопрос. Теперь уже я был задет за живое и нахмурился. Это замечательно – излагать теорию об отношениях теории и практики, но оправдается ли она на практике? (Нет ли здесь чего-то вроде рекурсии?)

У меня плохо работает творческая мысль, когда я стою перед публикой. Наверное, весь этот адреналин, необходимый для тонуса при публичном выступлении, плохо влияет на способность мозга к творчеству под взглядами десятка или сотни слушателей. И когда тот слушатель задал мне

сложный вопрос, я пробормотал что-то о творческом проектировании и графических интерфейсах пользователя как областях, где практика идет впереди теории, затем поспешно глянул на часы, посетовал на недостаток времени и продолжил выступление по заранее подготовленному материалу.

Уединившись в своем кабинете и дождавшись, пока адреналин вернется к норме, я снова задумался над тем вопросом. Действительно – где же практика по-прежнему опережает теорию? Некоторые результаты моих размышлений приведены ниже.

Проектирование программ

Люди проектируют программы уже больше полувека. Многие из выполненных проектов оказались успешными, иногда чрезвычайно. Эти успешные проекты – результат работы творческого ума над весьма сложными проблемами – вспомните слова Фреда Брукса о том, что программирование «наиболее сложный из всех известных видов человеческой деятельности». Таким образом, есть масса свидетельств в пользу того, что практика проектирования процветает.

А что можно сказать о теории? Возьмите любой учебник по проектированию программного обеспечения – что вы там найдете? Несколько методологических советов и несколько советов по представлению. Надо полагать, стоит последовать этим методическим указаниям, как у вас появятся мысли, достойные регистрации предлагаемыми способами. Пожалуй, этого недостаточно. От методологии до представления лежит долгий творческий путь, о котором в учебнике ничего не сказано.

Кроме того, задумайтесь: когда заговорили о методологиях и представлении в программировании? Не в 1970-х ли? А когда на практике стали проектировать реальные рабочие программы? Кажется, в 1950-х? К каким временам ни обратись – историческим или нынешним, – в этой области практика ведет теорию, сохраняя лидерство и сейчас.

Думаю, однако, следует признать, что теория упорно стремится догнать практику. Исследования когнитивного проектирования, выполненные командами Билла Кертиса (Bill Curtis) и Эллиота Солоуэя (Elliot Soloway), создали хорошие условия для того, чтобы теория сравнялась с практикой в творческом проектировании. Но эти результаты получены в 1980-х и не получили широкого признания в новейших учебниках и теоретических работах. Я бы сказал, что пока практика проектирования программ все еще опережает теорию. (Во второй части книги мы вернемся к этой теме.)

Сопровождение программ

На своей первой работе в компьютерной области – больше пятидесяти лет назад – я занимался программным сопровождением. Опыта в этом деле у меня тогда не было, но сначала мне повезло – программы, которые я должен был сопровождать, были написаны классными специалистами, а за несколько лет мое мастерство существенно выросло.

Чем же компьютерные науки и программный инжиниринг могли облегчить мне в те времена задачу сопровождения? Абсолютно ничем! На самом деле, тогда еще не было компьютерных наук и программного инжиниринга в академически-теоретическом смысле. Эти дисциплины удостоились допуска в аудитории престижных университетов лишь спустя больше десятка лет.

Но есть ли связь между этой древней историей и сегодняшним днем? Думаю, есть, и причины ее весьма печальны. До самого недавнего времени теория программирования пренебрегала сопровождением. Она не просто отставала от практики, но и не собиралась ее догонять! Литература по программированию демонстрировала поразительную скудость теоретических изысканий по сопровождению. Вспоминается выступление одного теоретика на компьютерной конференции, которое свидетельствовало о явном отставании от практики и только выиграло бы, разберись он в достигнутых ей результатах! (Эту историю я рассказывал в начале главы.)

Сейчас положение начинает меняться. И все же печально, что число мест, где произошли эти изменения, таких как Даремский университет в Англии, университеты Западной Флориды, Аризоны, Де Пол и ряд других, можно пересчитать по пальцам одной – максимум, двух рук. В мире науки по-прежнему отсутствует серьезный интерес к сопровождению программного обеспечения.

При этом существует хорошо известный теоретический подход, позволяющий теории – в любой области – догнать ушедшую вперед практику. Он называется «эмпирическим исследованием» и заключается в изучении лучших практических приемов с тем, чтобы развить теорию. Но теоретики программирования до сих пор не проявляли интереса к этому очень эффективному начальному методу, и в результате можно прямо сказать, что на момент публикации этой статьи мало что подтверждает большие успехи теории в попытке приблизиться к уровню, достигнутому практикой сопровождения программного обеспечения.

Интерфейс пользователя

Это прекрасный пример области, где теория и практика движутся ноздря в ноздрю, и есть все основания полагать, что теория вырывается вперед.

В то же время, это интересный пример того, что происходит, когда теория догоняет практику.

Сегодня каждый интересующийся компьютерами школьник знает о новаторской работе в области интерфейсов пользователя, проделанной несколько десятилетий назад в Xerox PARC. Как же нам расценивать эту работу? Как теоретическую? (Она была сделана в производственных условиях.) Или как практическую? (Ее проделали теоретики, но за исключением принадлежности к производству они обладали всеми атрибутами ученых.) По-моему, честнее признать эту работу сплавом теории и практики.

Что же было потом? Большая часть работы влилась в производство, где Apple превратила ее в отраслевой стандарт, сделав основой своего длительного экономического успеха и продолжив разрабатывать концепции Xerox PARC. Конечно, теория продолжает развиваться, но, по-видимому, не будет ошибкой сказать, что огромный скачок в теории и практике, совершенный много лет назад, до сих пор остался непревзойденным ни в том, ни в другом направлении.

Программирование больших систем

Практическое программирование всегда казалось программированием больших задач. То есть у каждого поколения практиков было ощущение, что создаваемые ими программы требуют большого напряжения умственных способностей и работают на пределе возможностей аппаратной части.

И, конечно, каждое поколение практиков оказывалось поверженным в прах стремительно растущей сложностью задач, решаемых новым поколением. С трудом верится – но опровергнуть эти цифры я не могу, – что в 1990-м году программные системы были в среднем в 50 раз крупнее, чем в 1980-м [Dekleva 1991]. Это ошеломляющее свидетельство скорости изменений в практике программирования.

А что в это время происходило в мире теории? В большинстве случаев теория по-прежнему развивалась путем ограниченного изучения небольших проектов. Практики в насмешку называют такие проекты «игрушечными», но я не стану их так унижать, а буду называть «малыми исследованиями». Есть ряд противоположных примеров, крупных исследований – здесь мне приходят на ум работы отдельных личностей, таких как Вик Базили (Vic Basili) в Мэрилендском университете и Крис Кемерер (Chris Kemerer) в Питтсбургском университете, а также несколько важных организаций, таких как Software Engineering Institute, Fraunhofer (в Германии и США), NICTA (в Австралии) и Simula (в Норвегии). Но в большинстве случаев теоретики заняты тем, что пытаются распространить результаты малых исследований на большие проекты. А с тех пор как более четверти века назад появился термин «программный инжиниринг» (soft-

ware engineering), большинству практиков известно, что такие попытки безуспешны.

Здесь, конечно, перед теорией встает сложная задача. Крупномасштабные исследования дороги, у большинства исследователей таких денег нет. то есть понятно, почему теоретический мир не дорос до потребностей практики.

При этом некоторые толстокожие теоретики посмеиваются над приверженностью практики старым технологиям и идеям. Иногда я спрашиваю себя: а что если бы практики программирования отпускали в адрес теоретиков аналогичные шутки? Не думаю, что такие взаимоотношения пошли бы на пользу нашему делу, но, как говорится, что соус для гусыни, то соус и для гусака.

Пора миру теории программирования дозреть до «профессорских исследований», как говорит Вик Базили. Что он имеет в виду? Я толкую его слова следующим образом. Новые преподаватели склонны проводить исследования, в которых глубоко изучают мелкие детали своей дисциплины. Такая работа продолжает то, чем они занимались в аспирантуре, и необходима для зачисления в штат. Но наступает время, когда кто-то – желательно, уже взявший барьер принятия в штат, – должен более широко заняться изучением основных проблем своей области. В идеале это профессора и другие старшие по должности.

Кто проводит исследования, как профессор? Помимо тех, кого я уже перечислил, интересно ознакомиться с трудами Мэри Шоу (Mary Shaw) из Университета Карнеги-Меллона, которая изучала происхождение и эволюцию разных инженерных дисциплин, чтобы понять эволюцию программного инжиниринга. По ее наблюдениям, такой метод многое дает для понимания нашей отрасли.

Более энергичное стремление к крупномасштабным и/или «профессорским» исследованиям, поддержанное соответствующими финансовыми вливаниями, смогло бы придать желаемую динамику исследованиям в этой области. Может быть, при достаточной поддержке они смогли бы даже подтянуться до практики крупных проектов.

Моделирование и эмулирование

Если вам доводилось участвовать в крупном проекте по разработке программного обеспечения реального времени, вы должны представлять себе, какую большую роль в такой работе играют моделирование и эмулирование. Разработка моделей и эмулирование позволяют:

1. Проанализировать концепцию системы, прежде чем приступать к ее созданию.

2. Исследовать проектные решения и проверить их жизнеспособность.
3. Позволить программному обеспечению, разрабатываемому для целевого компьютера, выполняться на компьютере разработчика с другим набором инструкций (ввиду недоступности целевого компьютера или недостаточной поддержки им средств отладки).
4. Подменить реальную среду тестирования на ранних этапах тестирования среды.
5. Создать подходящие тестовые оракулы для проверки результатов работы готовой системы во время системного тестирования.

Это большой объем моделирования и эмулирования. Как видите, технология занимает важное место в мире программирования систем реального времени.

А как тут обстоит с теорией? По моим наблюдениям, не очень. Например, судя по теоретическим курсам в учебном расписании, по этой теме практически ничего нет. В то же время на практике системы эмулирования набора инструкций, среды и так далее, о чем мы говорили, уже десятки лет являются совершенно необходимыми средствами для работы.

Конечно, моделирование и эмулирование интенсивно применяются в некоторых пограничных областях теоретической компьютерной науки, особенно относящихся к созданию приложений, но главные направления теоретических исследований их не затрагивают. Если моделирование и эмулирование и попадают в учебный план, то обычно не в тех колледжах, где преподаются компьютерные науки или программный инжиниринг.

Это то, что я называю «провалом», сравнивая теорию с практикой: предмет на практике существует, а в теории – нет. А когда в теории есть провалы, ей придется очень сильно потрудиться, чтобы хоть как-то приблизиться к практике.

Метрики

В этой области произошло раздвоение. Методы измерения работы программиста, предлагаемые теоретиками и практиками, это два корабля, плывущие в противоположных направлениях.

Возьмем, к примеру, энциклопедическую теоретическую работу по метрикам Зусе (Zuse) и сравним с практическими книгами Грейди (Grady). Или сравним работу по метрикам сложности Хэлстеда (Halstead) и Маккейба (McCabe) с работой по метрикам качества, финансировавшейся Rome Air Development Center (и отраженной во многих документах Национальной службы технической информации NTIS). Такое впечатление, что теоретики и практики почти прекратили общаться между собой.

Мне посчастливилось, будучи редактором журнала по программированию, прочесть комментарии анонимного теоретика-рецензента к статье о метриках, написанной одним теоретиком. Смысл их, в основном, сводился к следующему: зачем брать на себя труд предлагать новые непроверенные теоретические метрики, когда: а) в теории и так полно непроверенных метрик и б) практика отказалась от всех этих непроверенных теорий и быстро движется в другом направлении? Лично я считаю, что рецензент, скрывший свое имя, чтобы его не распяли коллеги-теоретики, был одним из немногих, кто понял, что происходит с метриками в программировании.

Конечно, некоторое перекрытие между теорией и практикой есть. Многие организации, занимающиеся сопровождением программного обеспечения, пользуются коммерческими средствами CASE для вычисления разных теоретических метрик вроде предложенных Хэлстедом и Маккейбом, утверждая, что с их помощью очень удобно выявлять участки сложного кода, особенно при планировании модернизации.

Но есть и другая необычная партия в мире метрик. Это многочисленные теоретики и практики, которые, исследовав разнообразные метрики сложности, сочли их сомнительными с точки зрения теоретической обоснованности и малополезными в практическом применении.

Подытоживая тему метрик, пожалуй, лучше всего сказать, что в этой области отношения между теорией и практикой очень неровные. Думаю все же, что практика очень медленно начинает вырабатывать некоторые практичные, полезные метрики, а теория все еще не может решить, есть ли вообще какая-то польза от ее результатов. Предположу, что после преодоления начальных затруднений в этой области возможно ее бурное развитие.

Вот я и ответил тому увлеченно озабоченному слушателю. По крайней мере, в таких областях, как проектирование и сопровождение программ, интерфейсы пользователя, программирование больших систем, моделирование и эмулирование, а также метрики, практика программирования опережает теорию.

Все мы люди, и, конечно, мне жаль, что я не смог так же ответить во время лекции. Но если тому слушателю случится натолкнуться на эту статью, он, возможно, вспомнит свой вопрос и удовлетворится запоздалым ответом на него.

Ссылки

DEKLEVA 1991 – Article in *Software Maintenance News,* February 1991, reporting on a survey conducted by Sasa Dekleva.

Глава 8 | Наука и производство

Введение
8.1. Дихотомия интересного/полезного
8.2. Дихотомия индивидуального/группового
8.3. Две фразы из поп-культуры
8.4. О понимании, согласии... и формальных методах
8.5. Структурированное исследование? (Немного лукавое предложение)
8.6. Плавающее соотношение передача/прием
8.7. Заседание комиссии, не ориентированной на цель
8.8. Точность и актуальность

Введение

Наши [японские] дети вымуштрованы, и нашу школьную систему пронизывает стремление... к единообразию... Отклонения от норм [едва допустимы]. Творчество наказуемо... Вопрос «почему» интерпретируется не как здоровое проявление любопытства, но как личное оскорбление учителя и сомнение в его честности... Интеллектуальная мощь Японии ограничена: страна упорно работает, но не хочет упорно думать.

Хисао Ямада

«The Discipline of Curiosity»,
Jenny Groen, Eefke Smit, and Juurd Eijsvoogel,
Elsevier Science, 1990

Между Востоком и Западом сложилось странное разделение труда – молчаливое соглашение, при котором научной работой в основном занимаются Соединенные Штаты и Европа, а Япония превращает сделанные там открытия в рыночные продукты.

Гораций Фриленд Джадсон

«The Search for Solutions»,
Holt, Rinehart and Winston, 1980, приводится
с разрешения of Henry Holt and Company, Inc.

Чем отличается эта глава, сравнивающая академическую сферу и производство, от предыдущей, где сравнивались теория и практика?

В предыдущей главе мы изучали интеллектуальные различия между миром увитых плющом стен и миром, которым правит нажива. В этой главе мы рассмотрим различия в их поведении.

Разумеется, теоретические изыскания проводятся по большей части в университетах, а почти вся практическая работа происходит на производстве. Однако даже такие окончательные формулировки заслуживают уточнений, которыми я их и снабдил. Я занимался промышленными «исследованиями и разработками» (которые назывались R&D[1] и обычно содержали гораздо больше R, чем D, так что в итоге оказывались весьма

[1] То есть Research and development – исследования и разработки. – *Прим. перев.*

близки к той академической теоретической работе) и академическими практическими проектами (в основном руководя работой аспирантов, для которых задача произвести нечто имеющее успех на рынке была вторичной).

Таким образом, есть логика в том, чтобы различать теорию/практику и науку/производство. Возможны различные варианты того, кто, чем и где занимается.

Но гораздо более интересно, как мне кажется, посмотреть, чем отличаются те люди, которые работают в науке и на производстве. Моя карьера идеально подходит для исследования этого вопроса. Проработав в промышленности почти 30 лет и занимаясь практически всеми видами работ, которые там существуют (разработка программ, сопровождение, управление, R&D), я еще около десяти лет в различных качествах провел в научных кругах. Я был штатным преподавателем и работал по контракту. Я занимался исследованиями и печатался в неплохих журналах. Я работал в попечительских советах и разрабатывал учебные планы, а также занимался всеми мелкими делами, связанными с учебным процессом. Короче, моя карьера охватывает оба эти весьма различных между собой мира.

Лакмусовой бумажкой такого моего биполярного воспитания стало годичное пребывание в Институте программирования (Software Engineering Institute). Не знаю, что вы думаете о SEI, но по моему мнению, он всегда являл собой странный и не вполне удачный сплав двух миров – производства и науки.

Ученые руководили проектами, оказавшими значительное влияние на программную индустрию, – или принимали в них участие. Производственники предоставляли данные для проектов, имевших, в основном, академический интерес. Различные подразделения SEI укомплектовывались людьми по-преимуществу из того или другого лагеря. Карьера многих производственников была связана с исследовательской частью R&D, что делало их производственниками лишь частично.

Почему я считаю, что сплав был не вполне успешным? Потому что производственникам приходится прилагать усилия, чтобы быть услышанными в SEI. Потому что настоящие практики обычно терпят неудачу в сравнении с исследователями от производства. Потому что представители академических кругов проявляют такое пренебрежение к производственникам, что последним трудно в достаточной мере влиять на курс, выбираемый SEI. Если культурологическое исследование компьютерной области когда-либо займется сравнением поведенческих и интеллектуальных аспектов производства и науки, а также оценкой успешности их взаимодействия, то SEI представит собой идеальный плавильный котел двух культур для проведения исследования, если не считать, что в этом пла-

вильном котле обнаружатся непереваренные куски, культурная принадлежность которых никогда по-настоящему не давала им раствориться в едином целом.

А пока такого исследования нет, вам предлагается набор статей, представляющих мой посильный вклад в дело компьютерной культурной антропологии.

Прежде чем двинуться дальше, отдадим долг определениям:

> «Производство», как сказано в моем словаре, это «выработка, изготовление товаров; отрасль, вид промышленности». (Еще раз обратите внимание, насколько философия фабричных труб укоренилась в словаре!)
>
> Напротив, «наука» относится к обучению – в школе, колледже или университете; «академический» – «ученый, а не технический или практический; представляющий чисто теоретический интерес без практического применения».

В этих определениях контраст виден резче, чем у остальных пар, которыми мы занимались в этой части.

Перейдем к делу...

8.1. Дихотомия интересного/полезного

Этические нормы должны быть у каждого.

Они дают человеку мерило, мотив и даже энтузиазм.

Однако с этикой связана одна проблема. Природа этических норм такова, что люди редко подвергают их сомнению. Какой бы этики вы ни придерживались, едва ли вы готовы допустить, что она неправильна.

Но даже в этой неколебимости этики есть как достоинства, так и недостатки. Хорошо, что благодаря этой устойчивости отношения между людьми приобретают надежность и предсказуемость. Плохо, потому что этические вопросы задевают за живое и служат источником раздоров.

Но хватит философии. Как этика связана с программным обеспечением?

Взглянем на две сферы программирования – академическую и промышленную, – и эта связь вскоре станет заметна.

В академическом мире исследователь занимается тем, что ему интересно. Это положительно оценивается его коллегами.

На производстве исследователь делает то, что считает полезным. Это положительно оценивается его руководством.

Разделение предметов исследований на интересные и полезные не столь безобидно. Академический ученый с некоторым презрением относится к вещам, которые всего лишь полезны. Работник промышленности с некоторым презрением относится к вещам, которые всего лишь интересны. Таким образом, каждая сторона оценивает базовые цели другой.

Фактически, стремление к интересной работе входит в этические нормы академического ученого, а стремление к полезной работе входит в этические нормы работника промышленности. И эти нормы настолько сильны, что почти не допускают компромисса. Приверженцы одного этического кодекса ставят под сомнение разумность и даже нравственность приверженцев другого.

Можно ли привести менее философские примеры этой дихотомии? Пожалуйста, сколько угодно.

Интересно ли доказывать корректность? Да. Есть ли от этого польза? Нет.

Интересно ли тестировать? Нет. Есть ли от этого польза? Да.

Интересно ли заниматься языками требований? Да. Есть ли от этого польза? Меньше, чем хотелось бы.

Интересны ли обзоры требований? Главным образом, нет. Есть ли от них польза? Да.

Интересно ли символическое выполнение? Да. Есть ли от него польза? Небольшая.

Интересно ли рецензирование кода коллегами? Нет. Есть ли от него польза? Еще какая!

Эта дихотомия встречается повсюду, и она вполне реальна.

И очень плохо, что она есть. Потому что исследователь, который занят, в основном, интересными проблемами, мало общается – если общается вообще – с исследователем, занимающимся главным образом темами, от которых есть польза. Взаимное пренебрежение – плохая основа для обмена знаниями.

Дихотомия только обостряется, если исследователь берется за интересную задачу, по ошибке сочтя ее полезной, или за полезную задачу, по ошибке сочтя ее интересной. Наведение мостов должно быть подлинным, а не вымышленным. Оно должно базироваться на знании, а не на невежестве.

Пора представителям академических кругов и производства пересмотреть свои этические нормы. Вполне допустимо работать над интересными задачами. Вполне допустимо работать над полезными задачами. Недопустимо выдавать одно за другое. Недопустимо пренебрежительно относиться к тем, кто сделал другой выбор.

Есть замечательное понятие, которое у нас в крови и которое я называю «местным патриотизмом». Мы всегда знаем, что школа, которую мы окончили, – лучшая в городе. И наш вуз был лучшим в штате. И наш город, район, область – лучшие в стране.

Наша религия – лучшая религия в мире. И наши моральные нормы – лучшие.

Местный патриотизм полезен. Но он также бессмыслен. Вероятность того, что моя школа, вуз, город, район или область действительно лучшие, равна нулю!

Взаимопонимание – вот что поможет преодолеть нелепый местный патриотизм. Оно же – путь к преодолению этических конфликтов.

Этические нормы, как уже говорилось, должны быть у каждого. И каждый, согласитесь, должен соединить эти нормы с изрядной долей взаимопонимания.

8.2. Дихотомия индивидуального/группового

Пограничный слой между наукой и производством – это место, где возникает турбулентность.

Есть хорошо известные виды турбулентности, скажем, когда основным стимулом для студента становится не учение, а обогащение.

Другие виды известны хуже. Здесь я и хочу рассказать об одной такой менее широко известной области.

В академическом мире студент действует как лицо, конкурирующее с другим лицами, а групповая деятельность часто воспринимается как попытка мошенничества.

В мире производства наемный работник действует как член команды, и групповая деятельность имеет преобладающее значение.

В академическом мире отдельной личности воздается хвала за ее успехи.

В мире производства роль отдельной личности часто теряется в безвестности.

В академическом мире студенты, пользующиеся чужими работами, воспринимаются в лучшем случае с подозрением.

В мире производства использование чужой работы – просто проявление здравого смысла. (Противоположный вариант с известным презрением называют «изобретением колеса».)

В академическом мире списывание осуждается.

В промышленном мире делать новые документы путем копирования частей старых считается правильным и разумным.

Переход от неправильного к правильному происходит внезапно. Переход от конкуренции к сотрудничеству оказывается неожиданным. Переход от личной значимости к безымянности бывает болезненным.

С любых сторон переход оказывается... турбулентным. В пограничных слоях так бывает всегда!

8.3. Две фразы из поп-культуры

Теоретики иногда говорят, что практики слишком мало думают, прежде чем взяться за дело. В поп-культуре это обозначается фразой:

> Внимание! Огонь! Целься!

Уязвленные практики нашли аналогичное высказывание для теоретиков, что простительно. Принимая во внимание склонность теоретиков указывать на недостатки практиков, расхожей фразой могла стать такая:

> Внимание! Обругай! Огонь!

8.4. О понимании, согласии... и формальных методах

Прошлой зимой на компьютерной конференции я свалял дурака. Я стал спорить с докладчиком. То есть сделал то, что терпеть не могу в других.

Что случилось? Что меня так задело?

Докладчик излагал некую концепцию. Воодушевясь представлением этой концепции, он, на волне возбуждения, выразил неудовольствие тем, что не все соглашались с его мыслями, и фактически стал осуждать несогласных. Поскольку я был с его идеями не согласен, во мне росло возмущение. Наконец, я не выдержал. Я выступил и постарался объяснить, что хотя мысль докладчика понятна, многие могут с ней не согласиться.

Чем больше я говорил, тем старательней докладчик объяснял свою идею, и чем больше он объяснял, тем сильнее я выражал несогласие с ней. Мы увязли в безвыходной ситуации, в конце концов придя в ярость. Другие участники и докладчики замолкли. Воцарилась атмосфера неловкости. По окончании заседания я крадучись покинул зал, жалея, что вообще в нем оказался.

Размышляя впоследствии о случившемся, я пришел к выводу, что мы с докладчиком действовали каждый в рамках своей модели общения.

В его модели достаточно было, чтобы я *понял* идею, которую он излагал. Если я не соглашался с его идеей, значит, я ее не понял. Если я не понял, следовало еще раз объяснить идею. Если и после этого я ее не понял, то потому что я глуп или невнимательно слушал.

В моей же модели общения было два разных компонента. Первая – это понимание, над чем и трудился докладчик. А вторая, совершенно независимая от понимания, – согласие. Мне была *понятна* излагаемая им концепция – просто я не *соглашался* с ней.

Размышляя над этой проблемой, я усмотрел ее связь с более глубоким социальным смыслом и пришел к выводу, что она представляет более общий интерес. (Этим я оправдываю описание на этих страницах перебранки, случившейся на какой-то конференции.) В начале жизни, когда наше развитие определяют авторитетные фигуры, «понять» и «согласиться» фактически синонимы. Кто-либо из родителей или учитель излагает идею, а ребенок или ученик соглашается с ней. Но по мере взросления ученика или ребенка разрыв между пониманием и согласием постепенно увеличивается. Родителю подростка или преподавателю студента уже недостаточно просто добиться понимания идеи. Они должны стремиться к ее принятию, а это гораздо более сложная и утомительная задача. Это связано с тем, что понимание достигается просто усвоением теоретического материала, а принятие связано с адаптацией идеи к имеющимся представлениям. И чем богаче система уже имеющихся у слушателя представлений, тем сложнее задача восприятия новой идеи.

(Типичная реакция авторитетной фигуры на раздвоение понимания и принятия во взрослеющем человеке, это, пожалуй, жалкий вопль всех родителей – «да потому что я твоя мать, вот почему!»)

Помню момент, когда во мне впервые проявилась эта разница. Я прочел – много десятилетий назад – «Взлет и падение Третьего рейха» Ширера. Размышляя над прочитанным, я вдруг осознал, что мне понятны представленные Ширером причины, по которым Гитлер уничтожал евреев и организовал холокост.

И я пришел в ужас. Потому что до того момента в моем личном опыте не было различия между пониманием и принятием, как и должно быть у ребенка или школьника. Принимал ли я данные Гитлером обоснования холокоста? Конечно, нет – я противился им всем своим существом. И в то же время я понимал их. В тот момент было совершенно необходимо разделить эти два понятия.

Теперь я хотел бы вернуться к некоторым недавним своим размышлениям и попробовать перевести этот разговор на системы и программы. В компьютерной литературе я часто наталкиваюсь на обсуждение каких-то новых идей (например, формальных методов), сопровождаемое серди-

тым замечанием, что в области практики эта идея не принята, *поскольку совершенно неизвестна*.

Полагаю, что достаточно хорошо разбираюсь в практике программирования, и по моим представлениям важно не столько то, известны какие-то идеи практикам или нет, сколько согласны ли они с ними. И часто случается – формальные методы служат здесь особенно хорошим примером, – что действительной причиной неприятия каких-то идей в мире практики оказывается не их непонимание, а их неодобрение.

Что происходит, когда теоретик осознает, что его новой идеей не пользуются? Он начинает усиленно разъяснять ее. Он хочет, чтобы ее поняли. А добившись понимания и обнаружив, что идеей все равно не пользуются, приходит в раздражение и начинает обвинять практиков в том, что они глупы или не умеют слушать, – знакомая картина? Разве в мире, где правят авторитеты, «понять» не значит «принять»?

Смысл, который я хотел передать, таков. Необходимо, чтобы авторитетные личности, так же как студенты и дети, отделяли понимание от согласия. Мера, в которой авторитетная личность позволяет студенту или школьнику разъединять эти два понятия, свидетельствует о степени уважения к созреванию этих студентов и школьников и достигнутой степени ее собственной зрелости.

Достигнув такого разделения, авторитетная личность может приступить к решению двух важных задач, которые на нее возлагаются:

1. Объяснить свою идею так, чтобы это было понятно.
2. Дополнительно изложить причины, по которым она должна быть принята.

Мне кажется, что в большинстве нынешних исследований в области программирования отсутствует такой компонент, как проверка. С моей точки зрения, для современных исследований характерно обилие определений, обилие объяснений, обилие пропаганды и крайний недостаток фактов в поддержку этой пропаганды. Оценочные исследования, направленные на поиск таких фактов, традиционно проводятся нашими более учеными собратьями, но в нашей области почти отсутствуют.

(Одно оценочное исследование по формальным методам, с которым я знаком, демонстрирует 9-процентный выигрыш в «общей стоимости разработки», достигаемый в результате применения языка формальных спецификаций Z [Ralston 1991]. В более широком мире поиска компромисса между стоимостью и выгодой, где формальным методам пришлось бы конкурировать с языками 4GL, и CASE-инструментами, и методологиями, и разукрупнением, чтобы привлечь внимание менеджеров и финансирование, технология, сулящая 9% прибыли и большие затраты на ос-

воение, в списке перспективных технологий заняла бы не самое высокое место.)

Возможно, почти полное отсутствие оценочных исследований частично обусловлено отсутствием разделения понимания и принятия у наших авторитетных личностей. Наверное, дело в плохо понятом и явно не сформулированном общем ощущении того, что задача исследователя ограничивается пониманием. Возможно, это связано с тем, что исследователь – авторитетная личность – еще не усвоил, что зрелость практика-программиста растет. Из-за этого он не видит необходимости снабжать свое объяснение обоснованием.

О различиях между академическим миром и производством я размышлял на протяжении многих лет. Например, я считаю, что в академическом мире хуже умеют слушать (обычные условия лекционного зала не способствуют развитию у лектора умения слушать) и менее ориентированы на цель (здесь работают, чтобы учиться, а не учатся, чтобы работать). А сейчас я добавил бы к этому свое убеждение, что практики более успешно справляются с разделением понимания и принятия, чем теоретики. (Ваша первая попытка заинтересовать высшее руководство новой идеей на основе одного лишь понимания станет также последней!)

Ладно, хватит уходить в сторону. Спрашиваете, каков итог того спора на конференции? Что ж, я действительно поставил себя в глупое положение, но не очень жалею, что так поступил!

Ссылки

Ralston 1991 – «Formal Methods: History, Practice, Trends and Prognosis», American Programmer, May 1991; T.J. Ralston and S.L. Gerhart.

8.5. Структурированное исследование? (Немного лукавое предложение)

Есть проблема, о которой почти все молчат, и пора пролить на нее свет. Я называю эту проблему «кризисом исследований в программировании».

Кризис исследований в программировании? Но *что* это за кризис?

Он заключается в том, что исследования не укладываются в отведенный бюджет и сроки, а их результаты ненадежны. И это действительно кризис. Слыхали ль *вы* когда-либо о проекте, реализованном с тщательным соблюдением бюджета и в предусмотренные сроки, результаты которого сразу можно было применить на практике?

Я готов не только сетовать по поводу этого кризиса. У меня есть конкретные предложения, как с ним бороться.

Думаю, для начала нужно структурировать наши исследования. Я даже за структурную революцию исследований.

Что я понимаю под структурированными исследованиями? Прежде всего, для проведения исследований необходим строгий, точный, упорядоченный и простой процесс. Никаких случайных приспособленческих goto. Никакого прежнего расхлябанного, неуправляемого, свободолюбивого расчета на счастливый случай. У нас будет жизненный цикл исследования, тщательное слежение за прохождением контрольных точек и полученными результатами и комплект создаваемой в ходе исследования документации, по которой руководство сможет отслеживать достигнутый прогресс.

Кроме того, введем для исследования метрики. Нам нужны средства для измерения как продуктивности, так и успешности исследовательских проектов. (Например, можно попробовать подсчитать количество человеко-часов на одну строку опубликованных научных статей.) Чтобы была возможность сравнить будущие исследования в этой новой парадигме с прежними анархическими попытками, начать сбор данных для метрик лучше прямо сейчас. Пожалуй, сбор метрических данных исследований станет первоочередной мерой для преодоления кризиса исследовательской работы.

Да, еще нам нужно определить процесс, применяемый в исследовании. Возможно, даже удастся определить модель исследовательского процесса и придумать язык процесса, позволяющий определить исследовательскую деятельность, и затем сравнивать с этой моделью каждый отдельный проект.

Что войдет в модель исследования? Во-первых, в ней должны быть все элементы жизненного цикла исследования. Мы опишем требования к проекту с помощью формального, точного, математического языка, что позволит ясно изложить их тому, кто финансирует работу. На самом деле, если язык окажется достаточно точен, можно подумать о том, чтобы в будущем автоматически генерировать результаты исследований из требований, описанных на этом точном языке.

Затем нам нужен проект исследования. Возможно, мы разработаем методологию проектирования исследований, «методологию рациональных исследований» (МРИ) – набор пошаговых операций и процессов для проведения исследования. А когда проект будет готов, можно составить его представление с помощью ряда структурных языков: ДПИ – язык диаграмм потоков идей, отображающих поток идей и управляющих ими процессов; ДСИ – язык диаграмм структур исследования, дающих иерархическое представление функций исследования; ЯМИ – язык моделирования исследований, позволяющих точно представить многочисленные

мелкие детали проекта. На самом деле, даже без автоматической генерации результатов исследования из спецификаций требований можно попытаться, применяя точные и подробные представления, завершить исследования трудами лаборантов, раз уж проект подробно описан.

А затем наступает черед реализации исследования. Если хорошо формализовать предшествующие процессы, реализация исследования не составит сложности. У нас есть папки проекта исследования (ППИ) – своего рода хранилище, куда помещается все, что в будущем может понадобиться для реализации исследования любому, кто только откроет эти ППИ, и мы будем вести структурированный сквозной контроль исследования (ССКИ), в ходе которого участники исследования станут изучать и оценивать результаты реализации исследования. Так мы подходим к процессу тестирования исследования.

Возможны два подхода к тестированию структурированных исследований. В традиционном подходе, конечно, используются структурированные или случайные (стохастические) наборы входных данных и прогоняются контрольные примеры для реализаций исследований с целью найти ошибки реализации. Либо проводится формальная верификация исследования с применением процессов математического доказательства, чтобы одновременно выявить ошибки и проверить корректность результатов. В любом случае процедуры тестирования выполняются в соответствии со структурированными планами, и их результаты отражаются в структурированных отчетах о тестировании.

Как насчет сопровождения? Такая проблема для исследований не стоит (либо она не отличается от процесса разработки исследования), поэтому мы не станем определять отдельный процесс сопровождения исследования. (Заметим, что только это позволит сократить бюджет исследования на 40–80%.)

Вот теперь, когда есть точный метод исследования, можно, наконец-то, контролировать этих исследователей. Можно оценить время, необходимое для проведения исследования, по количеству строк в статьях, опубликованных по его результатам. А имея хорошие оценки, использующие эти точные и структурированные методы, можно тщательнее контролировать исследование, что позволит разрешить кризис.

Есть еще одна сторона современных методов исследования, которая заслуживает внимания. Современные исследования очень эгоцентричны, и каждая работа и публикации по ней тесно связываются с престижем учреждения и отдельных лиц. Необходимо освободить исследования от таких личностных связей. Для этого следует регулярно проводить обсуждения согласно этапам жизненного цикла исследования, чтобы достигнутый прогресс был виден руководителям, заказчикам и коллегам. Кроме

того, нужен выборочный аудит, чтобы выявить ошибки в проектах, вызывающих тревогу, и усилить структурирующий процесс в тех проектах, где пытаются избежать его применения. В результате создается подход к исследованиям, основанный на командных целях и лишенный личных интересов.

Вот и все. При наличии четкой программы исследовательского процесса, высокой дисциплины исследователей, жизненного цикла исследования, разбитого на этапы, в конце которых можно точно измерять достигнутые результаты, а также рецензирования и аудита для обеспечения гласности процесса и метрик для оценки качества работы по завершении исследования, мы сможем начать контролировать эту неуловимую сферу деятельности.

Исследователи всех стран, ликуйте! Не за горами структурная революция, наступает эпоха неукоснительного применения строгих и формальных методов, и кризис в исследованиях скоро будет разрешен!

(Данное исследование проведено на средства Международного теологического общества научных исследований и прочих неконтролируемых явлений, которое автор благодарит за предоставленную возможность выполнить эту работу. В частности, важно отметить, что спонсоры не настаивали, чтобы само *это* исследование осуществлялось в соответствии с содержащимися в нем предложениями. Эти идеи, конечно, предназначены для *других* исследователей и программистов, а не для элиты, к которой относит себя автор.)

8.6. Плавающее соотношение передача/прием

В мой кабинет только что вошел один из лучших парней, какие только бывают. Я был рад его видеть, что ему тут же и сообщил. Однако подспудно у меня возникли иные ощущения. По неясной причине меня вдруг затошнило.

Лучший из парней был преподавателем факультета, и его кабинет располагался по соседству с моим. Он преподавал какую-то инженерную дисциплину, уже не помню, какую именно. Он был внимательным и рассудительным. Против чего же восстал мой желудок?

Я был тогда относительным новичком в академическом мире. После почти 30 лет на производстве мне было интересно заняться новым делом. Я любил свою работу – создание и сопровождение программ, но в тайне всегда подумывал о преподавании. И вот мечта сбылась. Я стал преподавателем кафедры программного инжиниринга в университете, который решился ввести в учебный план программирование. В нужное время я оказался в нужном месте, чему и радовался.

Я заговорил с лучшим из парней. Мы весело поболтали, обсудив несколько *злободневных тем*, а затем не спеша перешли к более серьезным вопросам, как это бывает с любителями поговорить. Наконец мы съехали на некую особенно глубокую тему, которая действительно могла поглотить нас целиком. Да, ради таких разговоров стоило влиться в академический мир!

И тут это началось – исподволь, так что я даже не сразу заметил. Равновесие в разговоре, столь выдержанное вначале, стало давать крен в сторону лучшего из парней. Постепенно его показатель «говорю/слушаю» все рос и рос, а мой, соответственно, снижался. Повторялась старая история. Мой гость любил говорить, а не слушать. «Я же тебя предупреждал», – ворчал мой желудок.

Приятная улыбка застыла у меня на лице, поскольку мне хотелось остаться любезным с этим лучшим из парней, но я быстро терял интерес к разговору, к этому человеку и вообще ко всему на свете. Начавшись как диалог, беседа превратилась в *лекцию*. И я перестал получать всякое удовольствие.

Итак, разговор шел, но уже без моего участия, и я задумался над тем, что произошло. Конечно, я и прежде встречал тех, кто говорит, не слушая, но в этом лучшем из парней я заметил то, с чем был знаком по опыту общения с другими работниками академической сферы – не такими хорошими, как этот, но все же достаточно приятными людьми. У меня возникло ощущение, что в академическом мире желание говорить, а не слушать, приняло болезненные размеры. То, что в мире производства встречается в виде отдельных случаев, в академических кругах носит характер свирепствующей эпидемии.

Как странно, подумал я, все еще храня на лице застывшую улыбку. Должно быть, что-то есть в самой обстановке университета, из-за чего люди разучиваются слушать. Что бы это могло быть?

Конечно, ответ достаточно очевиден. Чем зарабатывают себе на жизнь работники академического мира? Лекциями. Кто, в основном, слушает эти лекции? Студенты младших курсов. Немного поразмыслив над этими двумя фактами, я и обнаружил очевидный ответ. Академические работники не слушают, а говорят, потому что именно за это им платят, этому они учились и к этому лучше всего подготовлены. Мотивация слушать у них ослаблена, потому что те, с кем они обычно общаются (я приходил к выводу, что общение здесь идет в одном направлении), не имеют возможности разговаривать с лектором на равных. Вся повседневная деятельность академического работника побуждает его больше говорить, чем слушать.

Конечно, такое мироощущение вызывает определенные проблемы. Если человек плохо слушает, он редко знакомится с новыми точками зрения, а без этого мозг может атрофироваться. Что может быть страшнее для ученого? Что как не мозг является олицетворением ученого в мире? Ученый с атрофированным мозгом – это уже не ученый.

Я довольно долго размышлял об этом, пока тянулся разговор, и едва не упустил момент, когда он завершился.

«Мне пора», – сказал лучший из парней. И с искренней и теплой улыбкой попрощался. Вздрогнув, я возвратился к действительности.

Вздрогнул я дважды. В первый раз потому, что вернулся к активной жизни, где нужно было чем-то заниматься, в чем-то принимать участие. Пассивное восприятие отступило на какое-то время.

А во второй раз? Тут я просто испытал шок. Я сообразил, что во время нашей беседы почти выключил этого милейшего человека из своего восприятия, что пропустил мимо ушей последние две трети сказанного им, и смысл происшедшего вызвал во мне ужас. К тому моменту я уже достаточно долго был на преподавательской работе, и теперь осознал, что моя собственная способность слушать начинала атрофироваться. Я не научился достойно конкурировать в разговоре, но явно терял способность внимательно слушать. Мои мечтания по поводу говорю-и-не-слушаю свидетельствовали о начале моей трансформации в нежелательном направлении.

В тот день я принял решение бороться с такой тенденцией. Но достиг ли я здесь успеха? Чтобы выяснить это, нужно поговорить с кем-либо достаточно долго, а мне не удается. Что бы это значило?..

Постскриптум

В этой проблеме есть один аспект, особенно тревожный для меня. Я считаю – уже не один десяток лет, – что в области компьютеров и программирования между академическим миром и производством существует не просто недостаточный обмен информацией, но глубокая пропасть. Теоретики не слушают практиков, а практики – теоретиков. В чем проявляется это глубокое расхождение?

1. Профессиональные языки практиков и теоретиков плохо согласуются между собой. Пополняя свой жаргон во всяких учебниках или профессиональных публикациях, теоретик не может понять, почему практик не пользуется правильными терминами и правильным языком. Теоретики говорят «они неправы» по самым разным поводам, в том числе подразумевая «они изъясняются на некорректном языке». Иногда такое высказывание может быть справедливым, но отсутствие взаимопо-

нимания, порождающее подобные высказывания, может мешать содержательному общению.

2. Теоретики искренне уверены в ограниченности знаний практиков. Они рассуждают об адаптации презентаций до уровня практиков, говорят о «кризисе программирования» так, будто большинство практиков не понимают, что делают, и невысоко ценят умение разрабатывать и сопровождать системы, которое практики *реально* применяют. Интересно, каким образом, по мнению этих теоретиков, нам удалось, например, достичь таких успехов в космосе, если бы не способные и даже блестящие практики, успешно справляющиеся со своими задачами?
3. Похоже, теоретики не знают, чего они не знают. Они как-то уверовали в то, что прочтя все научные публикации в какой-то области, получаешь о ней полное представление. Им невдомек, что практика, порой – и даже довольно часто – опережая науку, отличается от нее в ряде существенных и важных аспектов.
4. Короче, часть академических работников – малая, но существенная – высокомерна и невежественна.

Конечно, это односторонняя картина существующего взаимного непонимания. Упрек заслужили и практики, которые не читают научной литературы и игнорируют мощные новые идеи.

И все же я иногда с интересом размышляю о том, в какой мере это глубокое расхождение обусловлено столь простой разницей между теоретиками и практиками: у теоретиков, как подсказывает мой опыт пребывания в обоих мирах, явно атрофировалось умение слушать. А это, разумеется, не сулит ничего хорошего в отношении преодоления существующей пропасти.

8.7. Заседание комиссии, не ориентированной на цель

На заседании комиссии факультета все засыпали. Это было уже третье заседание с одной и той же повесткой, и я начал подозревать, что мы никогда ни к чему не придем. В то время я даже не догадывался, насколько прав.

Совещание было посвящено выбору профилирующих дисциплин. Это важная тема. То, что мы выбирали, предстояло изучать всем студентам младших курсов. Независимо от их основной специальности. Трудно представить, чтобы какая-либо задача комиссии могла иметь большее значение.

Почему же мы так медленно подвигались вперед? Разумеется, велись обычные междоусобные войны за то, чтобы ввести ту или иную дисциплину в перечень профилирующих, или наоборот, вывести. (Тогда я еще не понимал, какая мука для преподавателя так называемые «вспомогательные курсы», то есть те, что читают студентам другой специализации. Учить студентов, которым навязали данный курс, было, как мне еще предстояло выяснить, худшей из всех возможных нагрузок преподавателя. Неудивительно, что преподаватели некоторых дисциплин не хотели, чтобы их предмет был включен в обязательный список.)

Однако главная проблема заключалась не в политике. Главной проблемой было, как мне кажется – и здесь мне приходится подыскивать нужные слова, десять лет спустя вспоминая свой опыт, – всеобщее равнодушие к тому, решим мы что-нибудь или нет. Нет сомнения, что участники совещания испытывали радость коллегиальности, но едва ли у кого-нибудь присутствовало ощущение значимости задачи.

Коллегиальность, но не общая задача. Вот такие не очень удачные слова я нашел для того, чтобы передать свою мысль. И я пришел к выводу, в справедливости которого – после многократного повторения одних и тех же ситуаций пока я работал в академическом мире – у меня нет сомнений: люди в академических кругах просто не ориентированы на достижение цели. Если производственное совещание созывается по определенному вопросу, разобрав который все расходятся, то академические совещания существуют ради самих себя.

В известном смысле я и сейчас считаю этот вывод верным. С другой стороны, думаю, он несправедлив. Дело в том, что на производстве и в академическом мире ставятся очень разные задачи. Как все уже, наверное, слышали, в академическом мире работают для того, чтобы учиться, а на производстве люди учатся для того, чтобы работать. В академическом мире занимаются тем, что интересно, а на производстве – тем, что полезно. И это не просто слова, а глубокое социальное различие, в значительной мере определяющее то, чем занимаются люди в каждой из этих сфер.

При этом ничто не может лучше мотивировать людей заниматься своим делом, чем стремление к общей цели. Мои коллеги по комиссии, занимающейся списком обязательных дисциплин, просто не считали конечный продукт достаточно важным, чтобы тратить усилия на его получение.

(Позднее я пришел к выводу, что проблема лежит еще глубже. Если производственники не согласны между собой по поводу методов и подходов, то абсолютная необходимость изготовить конечный продукт вынуждает их преодолеть свои разногласия и грести в одну сторону. Если разногласия по поводу методов и подходов возникают в академическом мире, мо-

жет не найтись ничего, что заставило бы прийти к общему решению. Если отсутствует стремление к общему продукту, трудно заставить группы с разными интересами работать совместно. В академическом мире споры, не приводящие ни к какому решению, могут протекать поразительно долго. И то, что главная цель вуза – такие задачи, как обучение студентов и научная работа, не помогает преодолеть разногласия, поскольку преподавание в целом рассматривается как неизбежное зло, а исследования часто ведутся индивидуально и потому не требуют группового согласия.)

Четвертое и пятое заседания комиссии по определению обязательных курсов лишь немного приблизили нас к поставленной цели, то есть к составлению согласованного списка. А затем – чудо из чудес – подошли летние каникулы. Разумеется, летом мы собираться не могли и предположительно должны были вернуться к своей задаче осенью.

А потом случилось кое-что интересное. За лето неведомым для нас всех образом список обязательных дисциплин все же был составлен. Осенью, когда мы вернулись, никаких совещаний уже не было. Это была радость, в которую верилось с трудом.

Но было ли это такой уж удачей? Задним числом я выяснил, что произошло в действительности: в течение лета руководство постаралось и оформило собственное представление о том, что надо включить в обязательный список, – возможно, это представление у него имелось с самого начала. Однако – и это важно – ни у кого из нашей комиссии не было ни малейшего желания возражать против этого списка, потому что мы не хотели больше никаких совещаний!

Так что же на самом деле тогда произошло? На языке передачи технологии и управления изменениями комиссия по составлению списка называлась бы инструментом вовлечения рядовых масс. Преподавателям, входившим в эту комиссию, была дана возможность принять участие в процессе и направить корабль к цели. То, что они не достигнут цели и в конце весеннего семестра оставят судно без руля, было предсказуемо. Но ориентированная на достижение конечной цели администрация приняла решение, по форме выглядевшее как привлечение преподавателей, благодаря чему они могли чувствовать свою причастность к делу. Администрация не только знала, что преподавательский состав не слишком ориентирован на достижение цели, но и использовала это в своих интересах.

Но, боюсь, мы отвлеклись. Мы занялись анализом последствий фундаментального различия между академической и практической сферами, хотя очерк посвящен самому различию, а не его последствиям. А различие заключается в ориентации на цель. Практики обычно сильно ориентированы на достижение цели; а «академики» либо вообще не ориентированы, либо их цели настолько необычны, что не сразу их и заметишь.

И это различие следует учитывать при любых попытках наведения мостов через пропасть между академическим и производственным мирами.

Постскриптум

Есть еще одно интересное проявление такого разного отношения к целям в академическом мире и на производстве. Как мы видели, на производстве люди ориентированы на продукт, поскольку в большинстве случаев именно продукт представляет собой конечную цель.

В последнее время в академическом мире бытует мнение, что путь к лучшему продукту лежит через процесс. Исходя из этого огромные усилия тратятся на то, чтобы усовершенствовать процесс. На этой предпосылке основана, например, вся модель зрелости процесса Института программирования (CMM).

Однако здесь следует проявлять осторожность. Сосредоточенность на процессе вместо продукта – естественная тенденция в академическом мире. Это вытекает из «работать, чтобы учиться» или «интересное лучше полезного». Стремление теоретиков убедить нас в том, что хороший процесс вернее всего ведет к хорошему продукту, нужно оценивать правильным образом, а именно, как естественное стремление группы не ориентированных на достижение конечной цели людей концентрировать внимание на предметах, интересных *им самим*, но, возможно, бесполезных для их коллег, более ориентированных на достижение целей.

Иными словами, нет ничего плохого в том, что практики будут заниматься процессом, если это не вредит продукту. Ориентированным на достижение конечной цели практикам никогда не следует об этом забывать.

8.8. Точность и актуальность

Классическую дилемму любой академической области, имеющей практическое применение, составляет проблема соотношения точности и актуальности, или что важнее – выполнить безупречно надежное исследование (точность) или исследование с большим потенциалом практической пользы (актуальность)? Особо важна она для практиков программирования, если они рассчитывают, что академическое исследование может оказаться в какой-то мере полезным для них.

Простому человеку такое противопоставление может показаться излишним: разве нельзя одновременно обеспечить точность и актуальность? Ответом на этот вопрос – к возможному удивлению тех, кто не занимается исследованиями (или даже не только их!), – будет решительное «нет». Для точного экспериментального исследования, к примеру, требуется хорошо контролируемая среда с узкими границами. Но чтобы исследование

оказалось полезным для практики, его нужно проводить в среде, как можно более близкой к реальной. А о реальном мире вряд ли можно сказать, что он хорошо контролируем и имеет узкие границы. (Как кто-то хорошо сказал, «реальность – это шайка гнусных фактов, убившая прекрасную теорию!»)

Возможно, нигде дилемма точности и актуальности не доставляет столько трудностей, как в области информационных систем (ИС). В принципе, ИС – академическая компьютерная дисциплина, занимающаяся прикладными компьютерными решениями бизнес-проблем, и потенциальная актуальность этой области должна быть оценена очень высоко. (Чего не скажешь о других областях, таких как «компьютерные науки» и, в меньшей степени, «программный инжиниринг», которые часто не интересуются практическим применением).

Эта проблема вызывала тревогу практически с первых академических исследований ИС, начавшихся несколько десятилетий назад. Периодически в академических кругах возникают дискуссии, касающиеся разных сторон этой дилеммы. Проблема представляется в нескольких разных обличьях. Что конкретно относится к области информационных систем? На каких главных вопросах сосредоточено ее внимание? Как она соотносится с компьютерными науками? Как она связана с другими базовыми дисциплинами? Опережают ли исследования ИС практику или отстают от нее? Если отстают, правильно ли такое положение? Это законные вопросы, и неспособность в течение многих лет ответить на них свидетельствует о некотором отсутствии порядка в этой области, что относится как к теоретикам, так и практикам.

Известны многочисленные попытки предложить решение этой дилеммы. Например, в одной недавней научной статье [Vessey and Rosemann 2005] предлагалось подвергнуть научные исследования «практической проверке», чтобы выяснить, соответствуют ли они таким практически важным критериям, как легкость понимания, актуальность, своевременность, применимость и публикация в местах, доступных для практиков.

Вспомните рассуждения о временном соотношении теории и практики, приведенные ранее в этой книге. Очевидно, академические исследователи ИС никак не могут согласиться с тем, что в некоторых случаях теория неизбежно оказывается в хвосте практики. В то же время практики ИС истомились в ожидании полезных для них научных результатов. Вот настоящая проблема, а не буря в стакане.

Эта проблема выдвинулась сейчас на передний план благодаря недавнему бурному обсуждению в Интернете относительной роли точности и актуальности в применении к исследованию ИС. Дискуссия велась на почтовом сервере академических исследователей ИС в оживленном, подчас

бурном режиме. Данный раздел посвящен итогам этой дискуссии с точки зрения практика.

Чтобы несколько упорядочить то, что по сути было собранием моих весьма личных взглядов и мнений, я выделил и привожу ниже несколько кратких цитат из этой дискуссии, определенным образом сгруппировав в надежде сделать их более содержательными для читателей, как практиков, так и теоретиков. (Обратите внимание, что авторы приведенных комментариев не указаны. Меня интересуют различные точки зрения, а не то, кому они принадлежат. Кроме того, хотя я и не расставил кавычки, фактически это буквальные цитаты из ряда сообщений электронной почты.)

Почему академические исследования ИС оказались в нынешнем состоянии?

Мы занимаемся исследованиями главным образом для того, чтобы удержаться на академической работе. В большинстве учебных заведений для включения в штат требуется определенный объем научных результатов... Это правила игры для желающих преуспеть в академической сфере...

У некоторых из моих учителей (к счастью, немногих) отсутствовал достаточный практический опыт, и они слабо представляли себе, что происходит в реальном мире.

Настоящий критерий исследований ИС должен звучать так: «Узнали мы что-нибудь новое об информационных технологиях с тех пор, как возникла наша область? Думаю, да».

Я думаю, что академические исследования в сравнении с производственными имеют следующие преимущества:

1. Направлены на проблемы, а не на коммерческие интересы.
2. Занимаются нерешенными задачами.
3. Занимаются проблемами, не интересующими производство по экономическим причинам.
4. Занимаются проблемами, в которых административный аспект важнее технического.
5. Изучают проблемы преподавания ИС.

В 1967 году Дуг Энгельбарт (Doug Engelbart) выполнил в Стэнфордском научно-исследовательском институте исследование, показавшееся практикам неактуальным. Лишь через пять лет им заинтересовалась Xerox. Исследование касались оконного интерфейса с использованием мыши. В 1967 году оно было неактуальным – сегодня оно очень даже актуально.

Проблема актуальности исследований для бизнес-практики... определяется культурой. Она не имела бы такого значения в какой-то другой культурной среде, отличной от западной (капиталистической) системы «свободного предпринимательства».

На каком основании актуальность наших исследований для сегодняшней бизнес-практики должна быть критерием оценки исследований ИС?

Взгляды тех, кто выражает интересы бизнес-сообщества, не должны служить основой для оценки актуальности работы. То, что хорошо для General Motors, не обязательно хорошо для всей страны.

Почему практика оказалась в нынешнем состоянии?

Работая на производстве, я не прочел ни одного научного журнала и, честно говоря, не помню случаев, чтобы кто-нибудь их читал.

Есть даже два класса журналов, каждый из которых отвечает интересам своей группы читателей – «Information Systems Research», «MIS Quarterly» и так далее – для фундаментальных исследований, и «Datamation», «CIO» и так далее – для прикладных исследований (любопытно, что этот академический автор не знал, что ряд указанных им журналов «прикладных исследований» а) на самом деле не пишет об исследованиях и б) прекратил свое существование! – *Прим. авт.*).

Не знаю, почему мои сотрудники не участвуют в научных конференциях. У некоторых из них есть ученые степени, и я сообщаю им обо всех приглашениях на интересные конференции.

Значительная часть того, что мне известно о практике, очень быстро устаревает.

Главное, что дает практический опыт, это понимание, насколько важны точность, тестирование и качество.

Разрыв между преподаванием/исследованиями/теорией и практикой

Существуют глубокие разногласия между академической сферой и производством. Частично эта проблема обусловлена недостаточным пониманием природы и цели исследований ИС.

Пожалуй, нет сколь-нибудь существенных академических открытий в ИС, а если и есть (немного), то, в основном, от бизнес-школ. Это подтверждается тем, что почти никто из деловых людей не тратит свое время на чтение академических журналов.

Доказательств сколь-нибудь существенного влияния исследований ИС на практику нет.

Исследования часто оказываются несвоевременными или неактуальными. Конференции запаздывают на пару лет, а обзоры в журналах появляются с задержкой на 1–3 года.

Мы явно отстаем от практики!!! Учебная дисциплина «информационно-управляющие системы» никуда не годится. Я не вижу никаких направлений, в которых мы бы опережали промышленность. Практики не уважают нас, потому что мы не уважаем их.

В других областях выдвигаются теории, которые неоднократно перепроверяются, потому что все пытаются их опровергнуть. В результате возникает теория, применимая к реальному миру. Насколько мне известно, в нашей сфере такого нет.

Если начать оценивать академические исследования по тому, чего от них ожидают деловые люди, результат окажется таким же плачевным, как если бы вы копали землю молотком или забивали гвозди лопатой.

У теоретиков и практиков разное мировоззрение. Теоретики предпочитают создавать общие концепции и идеи, а потом уточнять теории и концепции.

Нет оснований применять «исключающее ИЛИ» к точности/актуальности или теории/практике, нужно выбрать и то и другое.

Значительная часть исследований ИС опирается на эмпирические результаты, полученные на практике. Во многих случаях применения ИТ практика опережает теорию. Но есть и другие примеры.

В академических кругах часто не умеют установить связь с практиками, занятыми конкретным применением теории к конкретной задаче.

Старая поговорка гласит: теория без практики бесплодна, а практика без теории – слепа.

Консультантам очень хорошо платят за то, чтобы они обеспечили актуальность академических исследований для современной практики.

Компьютерные науки и информационные системы

Фундаментальные исследования совершенно необходимы, но их проводят в компьютерных науках (Computer Science), а не в ИС (IS). Если я хочу что-то узнать о новых технологиях, то стану читать «Communications of the ACM» или другие публикации в компьютерных науках, но едва ли в нашей собственной области.

Территория, захваченная к настоящему времени компьютерными науками и инжинирингом, такова, что практически исключает достижение ИС технического уровня, который представлял бы для нее угрозу.

Я не замечал, чтобы моих коллег из компьютерных наук беспокоили такие проблемы, как актуальность или влияние. Может быть, это как-то связано с творчеством. В творчестве (обычном занятии ученых в компьютерной области) проявляется ваша личность. Альтернативой ему выступает некая «паразитирующая» наука, которая занимается просто обзором и анализом и выдвигает странные гипотезы относительно того, как работают практики.

Мои коллеги (практики) все меньше интересуются конференциями IEEE и ACM.

Что делать?

Может быть, младшему преподавательскому составу и приходится совершать телодвижения, необходимые для зачисления в должность, но было бы неплохо, если бы их старшие коллеги стремились завоевать репутацию для себя лично и своей организации в целом за пределами башни из слоновой кости, в которой они находятся.

Почему бы нам не принять модель, существующую в медицине, где академический мир и практика объединены? Медицинские учебные заведения располагаются рядом с больницами или прямо в них, а большинство исследователей одновременно являются практиками или хотя бы действующими консультантами. Мы должны заставить крупные компании, такие как IBM, HP, Oracle, Microsoft и так далее, взять на себя роль «больниц» по отношению к нам.

Мне недавно довелось участвовать в конференции, где главными докладчиками были исключительно «академики», а фоном, в основном, практики. Меня поразили вопросы, которые практики задавали теоретикам: правильно ли мы действуем? как измерить успех? каково наше влияние?

Не пора ли подумать о стратегическом союзе с компьютерными науками?

Не следует ли потребовать, чтобы подготовка к получению докторской степени и курсовые работы старшекурсников обязательно включали в себя 3–6 часов «прикладных работ»? От студентов младших курсов мы требуем такой работы, но многие студенты старших курсов полностью ее избегают.

Соискатели докторской степени должны иметь некоторый опыт практической работы. Всем нам нужна периодическая переподготовка (до современного уровня практики, а не только науки).

Не пора ли ввести новый вид докторантуры – назовем его программой «ученый-практик», – чтобы привлечь лучших профессионалов среднего производственного звена и подготовить их до докторского уровня? Если нужно добиться «актуальности» наших исследовательских и образова-

тельных программ в области ИТ, разве не выиграют наши программы ИТ будучи дополненными такими «выкрестами»? И разве не выиграет производство от того, что наиболее талантливые его представители получат лучшую подготовку на докторском уровне и смогут обогатить практику результатами самых «актуальных» научных исследований?

Программы обмена с производством будут полезны всем – как опытным штатным преподавателям, так и практикам, соприкоснувшимся с академической средой.

Ссылки

Vessey and Rosemann 2005 – «Linking Theory and Practice: Performing a Reality Check on a Model of IS Success», working paper, Queensland University of Technology Center for Information Technology, 2005; Iris Vessey and Michael Rosemann.

Глава 9 | Забавность и серьезность

Введение
9.1. Разрываясь между забавностью и скукой
9.2. Open Source: возвращение забавности
9.3. Путешествие по странному проекту
9.4. Помогите найти!

Введение

Творчество, возможность его проявить, по-настоящему радует... Получая больше удовольствия от творчества, человек предлагает более эффективные решения.

Клер Арнольд,

глава Leapfrog Services, Inc., в «The Creative Edge», *Atlanta Business Chronicle*, 16 сентября 2005 года

Отсюда явно следует невозможность... увеличения размеров механизмов до невообразимых пределов,... постройки огромных судов, дворцов или храмов.

Галилей

цитируется по книге «The Search for Solutions» Горация Фриленда Джадсона, Holt, Rinehart and Winston, 1980, с разрешения Henry Holt and Company, Inc.

Пора действительно заняться делом. Все прежние разговоры о дисциплине, разумной достаточности, качестве и процессе были лишь разминкой. И только сейчас мы подходим к истине!

Программирование всегда было и остается интересным. На том оно и держится. Ты что-то сделал, и если все довольны, ты счастлив. Как говорится в рекламе пива, не нравится – найди себе лучше.

Послушайте ветерана. Я работаю с компьютерами с тех пор, как в далеких 1950-х началась их эпоха. В те времена компьютерами занимались исключительно ради удовольствия, ибо общество этого не поощряло.

В чем это выражалось? Нас считали чудаками. Собственной головой мы делали то, что все остальные делали руками: создавали некий продукт и заставляли его проделывать невообразимые вещи (в данном случае, с помощью ящика, набитого таинственной электроникой, в которой мало кто разбирался).

Не нужно путать нас тогдашних с нынешними хакерами. Хакеры делают обычные вещи весьма странными способами. А мы были пионерами, и необычным было то, что мы делали, но как мы это делали, никого не интересовало. Мы, пионеры, были странными из-за самой своей профессии;

хакеры странно себя ведут, но их профессия ни у кого не вызывает сомнений.

Помню, как летел в командировку (в 1970-х, из Альбукерка в Остин) и впервые моим соседом в самолете оказался человек, понимающий компьютерную терминологию. Я тогда уже лет двадцать занимался компьютерами, но мне еще ни разу не попадался кто-либо, способный связать пару слов на эту тему. А теперь вероятность встретить человека, который ничего не знает о компьютерах и не имеет о них собственного мнения, практически равна нулю.

Нас развлекала и эта странность. Мы действительно были пионерами и занимались тем, чего большинство людей не могло даже вообразить. Границы нашей деятельности были неощутимы. Объяснить не компьютерщику то, чем занимаешься, чаще всего было невозможно. Как это было забавно!

Но гораздо интереснее было программировать: думать, как применить какую-то новую команду самого нового компьютера, как переписать прошлогодние программы с использованием возможностей нового компьютера (мы занимались этим постоянно, потому что аппаратное обеспечение менялось быстрее, чем сегодня, хоть в это и трудно поверить), отлаживать программу, чувствуя себя сыщиком на пути к разгадке тайны, и видеть результат своих трудов, когда настоящий – может быть, даже правильный – ответ появляется на печатающем устройстве или терминале.

В значительной мере этот интерес сохранился и сегодня. Но теперь мы разбираемся не с новой командой процессора, а с функцией нового языка программирования. Нам не приходится регулярно переписывать старые программы (и это хорошо!). Но мы все так же испытываем радость погони при отладке и удовольствие от результата. Иногда я думаю: какой удачей было найти такое интересное дело, за которое еще и платят. Разве мы занимались этим только ради денег?

Но со временем что-то пошло не так, и я не могу сказать, что именно. Работа с компьютерами перестала доставлять прежнее удовольствие, о чем и говорится в заключительном разделе этой главы (и части). Может, я сам изменился и не могу больше сочетать легкость с продуктивностью? Или изменились условия работы, и программисты слишком озабочены немыслимыми ограничениями бюджета и сроков? Я не знаю ответа, хотя мне кажется, что верны оба предположения. Если у вас есть мысли по этому поводу, я с удовольствием их выслушаю.

Как бы там ни было, больше не стану отвлекать вас. Можно ли дать определение «забавного», «веселого»? Почему бы нет: «легкомысленное развлечение». (Представьте, что сами, не зная языка, пытаетесь что-то извлечь из *такого* определения! Легкомысленно, конечно.)

А как насчет «серьезного»? «Мрачный и задумчивый, не улыбающийся; нешуточный, не случайный или легкомысленный».

Тогда придется посмотреть, что значит «легкомысленный»: «бодрый, беззаботный; слишком небрежный, несерьезно относящийся к делу».

Эти слова слишком многозначны. Каким бы вы хотели быть – веселым или серьезным? Вот этим «не улыбающийся» многое сказано.

Но хватит играть словами. Приступим к теме.

9.1. Разрываясь между забавностью и скукой

О таком аспекте программирования, как забавность, слышишь достаточно редко. Конечно, кое-что об этом есть в литературе, посвященной Open Source, – в конце концов, если за программы не платят, то писать их стоит только для развлечения!

Но в остальных компьютерных публикациях забавность per se почти не обсуждается. Слово «fun» редко попадается в популярной компьютерной прессе. Его точно не встретишь в академической литературе. (Порой мне кажется, что теоретики специально стараются показать, что им не до забав. Как иначе объяснить скучнейший стиль большинства научных статей!)

Поэтому мне было приятно узнать, что мой друг Брюс Блюм (Bruce Blum) не только много размышлял о забавной стороне программирования, но и написал статью на эту тему [Blum 1991]. Он сразу согласился опубликовать ее в моем бюллетене (цитату из которого я привел выше).

Позвольте поделиться с вами некоторыми высказываниями Брюса о забавном.

В начале статьи он жалуется на то, как редко пополняется его папка *FUN*. Как бы вы ни искали забавы в своих занятиях программированием, говорит он, это редкая птица.

Где нас ждет развлечение? Брюс рассматривает это как в исторической перспективе, так и с точки зрения жизненного цикла.

Он рассуждает, какие этапы жизненного цикла могут быть забавными, а какие – нет. Дальше предлагается небольшая игра. Насколько вы согласитесь с мнением Брюса относительно развлекательности этапов жизненного цикла?

Первый этап жизненного цикла Брюс называет «соблазнением». Нельзя начать писать программу, пока кто-то не поддержит ваше решение ее написать. А соблазнение замыслом, по мнению Брюса, – почти чистое удовольствие. «Это решение задачи в самом свободном и увлекательном виде. Почти все возражения отвергаются мановением руки... Скуку вызыва-

ет только поиск беспосадочных рейсов, внесение последних поправок в рекламные материалы и общение с недалекими людьми». Здесь, по его мнению, развлечение и скука соотносятся как 100 к 0%.

Вторым этапом Брюс считает *составление требований*. Этот процесс он поровну делит между развлечением и скукой (50/50). Развлечение – разбираться в исходной задаче и ощущать прогресс в создании новой системы. Скука – корректно записывать все, ориентируясь при этом на то, что уже обещано. «Эта ранняя стадия может проходить очень легко», – заключает он.

Третий этап – *разработка общего проекта*. Здесь, по словам Брюса, «...мы видим нарастание скуки. Дав обещания, теперь мы должны придумать, как их выполнить. С ростом объема документации у нас остается все меньше свободы для принятия решения на месте. Источник нарастающей скуки (Брюс оценивает отношение забава/скука примерно как 40/60) очевиден. Нужно читать все эти документы, писать новые и – что угнетает больше всего – стараться не противоречить сказанному ранее».

На четвертом этапе – *детального проектирования* – все еще ухудшается, говорит Брюс (примерное соотношение 33/67). Это происходит потому, что вместо размахивания руками, как на предыдущем этапе, приходится придумывать способы, как реально воплотить полет фантазии. Заниматься конкретными деталями совершенно необходимо, но кто делает это с удовольствием?

В то же время, Брюс любит *программирование* – пятый этап. «Программировать, – пишет он, – по большей части, забавно. Вот почему это становится увлечением, вот почему [журналы по программированию] такие толстые и в них полно рекламы. Знаете, когда у меня все валится из рук, я просто сажусь за компьютер и пишу какую-нибудь программу. Это помогает мне поднять настроение». (Здесь Брюс оценивает отношение забава/скука как 75/25.)

Выше я предложил читателю самому оценить соотношение забавности и скуки. Думаю, именно этап программирования вызовет наибольшее несогласие. Как вы могли догадаться, я разделяю мнение Брюса относительно увлекательной стороны программирования. Но в разных учебных заведениях я встречал столько юных студентов, которые только и думают, как бы покончить с программированием, став руководителями тех, кто программирует. Меня, понятно, не перестает удивлять, что они выбрали компьютерную специальность.

Но я отвлекся. У Брюса есть еще пара этапов жизненного цикла, которым он дал оценку.

Шестым этапом он называет *тестирование*. Здесь, как отмечает Брюс, уровень забавности быстро скатывается к прежней отметке (33/67). Мы с ним явно по-разному смотрим на возможность получить удовольствие на этом этапе. Выше я уже описал этот этап как детективную историю, в которой нужно сопоставить все догадки и, в конце концов, найти и исправить все ошибки, которые, как ни трудно в это поверить, я случайно допустил в своем коде. Брюс смотрит на это дело иначе: «Честно говоря, во враждебной обстановке я занимаю оборонительную позицию». (Брюс рассматривает свои отношения программиста с официальными тестерами как враждебные. Но, в некотором смысле, компьютер, выполняющий мой фактический – и неправильный – код, тоже мой противник!) «Я уверен, что нашел бы [эти чертовы ошибки] сам, если бы не пришлось столько времени тратить [на выяснение отношений] при тестировании», – раздраженно заявляет он. И, чего скрывать, я разделяю его безрадостный взгляд на враждебность тестирования. Скажем, я жду, что мой код, как обычно, правильно заработает с первого раза (конечно, так бывает редко!), и тут появляется тестер, да и без него машина в очередной раз показывает мне, что я никуда не гожусь. Кому это понравится?!

Однако тестирование – это еще не все. Для цикла разработки это, возможно, окончание, но потом, когда разработка закончена, наступает (барабанная дробь!) этап *сопровождения*. Тут я слышу громкие возражения читателей, если я правильно понял, согласных с Брюсом. «Это почти чистая скука [0/100]», – говорит он. – Не думаю, чтобы кто-нибудь из программистов любил заниматься сопровождением. Иначе почему бы большинству новоиспеченных программистов поручались именно задачи сопровождения? И почему, иначе, среди людей, занимающихся сопровождением, самый высокий процент тех, кто «последним пришел, первым уволен»? (Потому что старые работники, как всем известно, боятся сопровождения как чумы.) (Кстати, я против того, чтобы для какой-то группы – за исключением бездельников – действовало правило «последним пришел, первым уволен», но по каким-то причинам находятся люди, например, женщины, для которых это справедливо. Заглянув в любую организацию, занимающуюся сопровождением программного обеспечения, вы обнаружите там поразительно большой процент женщин.)

Здесь я должен заметить, что лично я нахожу в сопровождении в некотором роде увлекательным. Правда, мне повезло: занимаясь сопровождением кода, написанного блестящими программистами, я многому научился и вырос профессионально. (Для пущего эффекта могу рассказать о том, как юный Билл Гейтс таскал листинги коммерческой операционной системы из мусорного ящика вычислительного центра своего университета, чтобы изучать по ним программирование.)

Выше я сказал, что Брюс изложил две разные точки зрения на соотношение интересного и скучного. С жизненным циклом мы разобрались. Теперь займемся историческим аспектом.

«Вначале, – пишет Брюс, – программирование занималось решением небольших задач... Жесткие требования ставились редко. Программисты работали с прототипами и с их помощью расширяли свои знания, создавая новые продукты. При этом они старались получать максимум удовольствия».

Но, считает Брюс, долго так продолжаться не могло. «По мере совершенствования технологии программирования и проектирования все более крупных систем выявлялись неотъемлемые проблемы такого индивидуального подхода. Команда из сотни программистов работоспособна лишь тогда, когда есть зафиксированные требования и проект. В результате программный инжиниринг постарался установить единообразный процесс, не интересуясь его увлекательностью».

«Очевидно, – продолжает Брюс, – что дисциплина, введенная на этом этапе развития программирования, действительно улучшила процесс. Была установлена своего рода математическая точность. Требования брались за аксиомы, а конструкции создавались как теоремы... Переход к дисциплинированному процессу оказался очень эффективным для больших и напряженных проектов».

Но всей этой дисциплины недостаточно. Есть задачи, для которых слишком строгий подход в виде аксиом и теорем не годится. Вопреки нашему желанию некоторые проблемы требуют большей свободы, чем допускают такие подходы.

«Проблема кроется в следующем, – пишет Брюс. – Процесс, оптимизированный для получения удовольствия, хорош лишь тогда, когда задачи малы, а качество определяется личными критериями. Процесс, оптимизированный для поэтапной разработки программного обеспечения, действует только при наличии жестких и устойчивых требований. Он непригоден, если требования выражены нечетко... Он также непригоден, если требования часто меняются. В таких случаях лучшими моделями для управления процессом разработки оказывается естественный интуитивный подход или контролируемая реакция».

Брюс написал это много лет назад, но его слова до сих пор производят на меня большое впечатление. Советую вам еще раз перечесть предыдущий абзац. Ведь еще в 1991 году Брюс написал о причинах, по которым все наши методы разработки программ работают хуже, чем нам бы того хотелось. В каком-то смысле он предугадал и Agile, и открытый исходный код, и прочие наши многолетние отчаянные попытки справиться с новы-

ми задачами, для которых не годится ни подход на основе увлекательности, ни подход поэтапной разработки.

Но пора взбодриться. «Вероятно, – заключает Брюс, – читатель хотел бы узнать, как же *надо* разрабатывать программное обеспечение. На самом деле, ответ прост. Нам нужна структурированная, объектно-ориентированная, использующая средства искусственного интеллекта парадигма прототипирования на базе формальных спецификаций и доказательств корректности, объединяющая основные функции композиции и декомпозиции в среде CASE с применением не зависящих от методологии методов и визуального программирования на Ada. Получив грант для этого исследования, я сразу продемонстрирую, как все это работает, – программирование действительно станет увлекательным занятием».

Хорошо сказано! По крайней мере, Брюс еще раз показал, что если программировать сейчас не так приятно, как раньше, то писать об этом попрежнему интересно!

P. S. Возможно, вас смутило слово «скука» в рассуждениях Брюса. Помоему, оно очень удачно, и я одобряю выбор Брюса. Но у этого слова есть и другой смысл. Брюс действительно много работал над созданием системы, которая «автоматизирует скучные стороны процесса программирования». Он назвал эту систему TEDIUM. Как и многие другие блестящие идеи в программировании, TEDIUM практически ничего не дала, за исключением нескольких книг Брюса о ней и, отчасти, внутреннего применения в Университете Джона Гопкинса, где Брюс работал до того, как несколько лет назад вышел на пенсию.

Ссылки

Blum 1991 – «Torn Between Fun and Tedium», *The Software Practitioner*, May 1991; Bruce I. Blum.

9.2. Open Source: возвращение забавности

Когда-то программы были бесплатными. Каких-то полвека назад программное обеспечение не продавалось – оно бесплатно прилагалось к компьютерному аппаратному обеспечению. И все программное обеспечение было открытым: если вы написали что-то полезное или интересное, это можно было поместить в общественное хранилище программных компонентов, вроде организации SHARE пользователей IBM, откуда любой программист мог взять его для своих нужд.

И программировать было интересно. Большинство программистов испытывали такое глубокое удовлетворение от творческого программирова-

ния, что занимались бы этим бесплатно, если бы, к счастью, не нашлись компании, которые стали за это неплохо платить.

Те времена миновали. Программы стали продуктом, стоящим денег – таких денег, что один предприниматель в области программирования стал самым богатым в мире человеком. Но что-то при этом пошло не так, потому что многие программисты стали находить свое занятие менее интересным. Огромные и сложные проекты, строгие и обязательные методологии, жесткие графики и суровый контроль со стороны руководства привели к снижению получаемого от работы удовольствия, которое на заре программирования было очень мощным стимулом.

Но это было тогда, а что теперь? Лет десять назад началось движение за то, чтобы вернуть в программирование понятия «бесплатно» и «интересно». И сегодня, когда готовится второе издание данной книги, движение Open Source, похоже, набрало скорость. Участвующие в нем программисты пишут программы в основном ради удовольствия.

Есть, конечно, и другие мотивы. Они пишут те программы, которыми хотят пользоваться. Они пишут программы, потому что хорошо умеют это делать. Они пишут программы в расчете, что те, возможно, будут приняты и высоко оценены другими программистами сообщества. Они пишут программы, потому что им не нравится нынешняя индустрия закрытого программного обеспечения, и в особенности Microsoft.

Но где-то ближе к вершине этого списка мотиваций лежит удовольствие от писания программ. Недавно даже было проведено исследование [Hann 2004], которое показало, что если рассмотреть все мотивы участия программистов в Open Source, то интерес (в исследовании он назван «развлекательной ценностью» (recreational value) – разве могут теоретики обойтись без пары длинных слов там, где достаточно одного короткого!) оказывается на втором месте в списке. Согласно этому исследованию мотивы участия программистов в Open Source таковы:

- 27% программистов участвуют в проектах ради практической ценности результата;
- 19% – ради той самой развлекательной ценности;
- 12% – в расчете, что это положительно скажется на их карьере.

Как видите, участие в проектах Open Source заметно связано с намерением получить удовольствие. (Кстати, на конференции доклад был отмечен за лучший выбор темы.)

Конечно, о движении Open Source можно сказать гораздо больше. Существует мнение, что в Open Source участвуют сливки программистского сообщества, они пишут самые надежные и безопасные программы, и Open Source принадлежит будущее. Но есть противники и опровергатели подоб-

ных мнений. Проповедников движения обвиняют в фанатизме, а противников – в непонимании предмета спора. В сражениях по поводу Open Source даже подняла свою жуткую голову политика, когда сторонники Open Source стали пытаться заставить некоторые правительства заменить их продуктами коммерческие программы, а компании-производители коммерческого ПО стараются отстранить Open Source от участия в проектах определенного рода. Временами все это смотрится довольно неприятно.

Если хотите получить более широкое представление о движении Open Source со всеми его достоинствами и недостатками, возьмите недавнюю книгу [Feller и др. 2005], представляющую собой избранные статьи об Open Source известных в компьютерной прессе авторов и «экспертов» Open Source. Если хотите быть в курсе событий этой быстро развивающейся области, читайте такие издания, как «Cutter IT Journal» или «Software Practitioner», регулярно публикующие статьи по специальным проблемам, которые к ней относятся. Если хотите подробно узнать, как участвовать в Open Source, но при этом еще и зарабатывать деньги (вот капиталист!), посмотрите статьи вроде [Bollinger 2004].

Если вам показалось, что предыдущий абзац демонстрирует нежелание заниматься этой темой и стремление поскорее завершить раздел, посвященный Open Source, то вы не ошиблись. О движении Open Source можно сказать много, но не здесь. В этой книге, посвященной творчеству в программировании, обсуждаются темы, имеющие хоть какую-нибудь связь с главным предметом. Думаю, не стоит считать Open Source более творческим способом программирования. Рискну утверждать, что здесь нет свидетельств как за, так и против. Этот раздел должен был показать, что благодаря Open Source к программированию отчасти вернулся интерес (э-э, «развлекательная ценность»), за что мы его здесь и благодарим, но не более того.

Ссылки

Bollinger 2004 – «How to Make Money Writing Open Source Software», *The Software Practitioner*, July 2004; Terry Bollinger.

Feller et al 2005 – *Making Sense of the Bazaar: Perspectives on Open Source and Free Software*, MIT Press, 2005; edited by Joseph Feller et al.

Hann 2004 – «Why Developers Participate in Open Source Software Projects», *Proceedings of the 25th International Conference on Information Systems*, Washington DC, Dec., 2004; Il-Horn Hann, Jeff Roberts, and Sandra A. Slaughter.

9.3. Путешествие по странному проекту

Статья Дуэйна Филипса (Dwayne Phillips) «Peculiar Project» (Странный проект) первоначально опубликована в «Cutter IT E-Mail Advisor» в декабре 2005 г.

Печатается с разрешения *dwayne.phillips.home.att.net*.

Peculiar, прилагательное – выходящий за рамки или отличный от обычного или ожидаемого, необычный гибрид.

Wordnet 2.0, Copyright Princeton University

Я работаю над странным проектом. Мне приходилось участвовать в странных проектах, но гораздо реже, чем в обычных. Однако на этот раз мне посчастливилось заметить странность этого проекта и порадоваться ей. Я выделил несколько характеристик странных проектов и выяснил, как сделать их нормой или хотя бы заставить случаться немного чаще.

Мой нынешний проект согласно приведенному определению «отличен от обычного или ожидаемого». Я считаю его странным, потому что все проблемы являются техническими, а все удовольствие связано с участниками проекта.

Технические проблемы и удовольствие от общения – отнюдь не норма, судя по моему профессиональному опыту. Обычно в проекте бывает несколько технических проблем, но все неприятности связаны с людьми. Люди ссорятся по поводу методов и процессов; они опаздывают на работу и уходят до конца рабочего дня; некоторые работают по 16 часов в день, устают и ворчат; кому-то надоедает проект, и он уходит, не сообщив, что он сделал, а чего не успел. Список связанных с людьми неприятностей можно продолжать почти до бесконечности.

И наоборот: большинство из тех, кто работает в моем теперешнем проекте, увлечены работой, счастливы и приятны в общении. Они не появляются и не исчезают из проекта с такой скоростью, что каждый месяц всем остальным приходится начинать все сначала. Если они уходят, то в хорошем настроении и в хорошо выбранный момент. В результате те, кто приходит им на смену, легко подхватывают то, чем они занимались, а всем остальным не приходится прекращать свою работу и учить новичков.

Работать над таким странным проектом приятно. Помнится, я назвал это удовольствием.

Честь организации такого странного проекта выпала не мне. Я горжусь тем, что развил в себе умение руководить работой и общаться с людьми, но по-прежнему многие из проектов, над которыми я работаю, – обычные. Этот проект заставил меня задуматься: почему доставляющий такое удовольствие проект необычен? Почему *большинство* наших проектов не могут так же радовать? Мне кажется, главная причина в том, что мы не рассчитываем, что наши проекты будут иными. Если мы не рассчитываем на работу с увлеченными людьми и дружескими отношениями, то и не стремимся создавать такие проекты.

Я внимательно изучил проекты, в которых участвовал. За 25 лет я только однажды составил список качеств тех людей, с которыми я хотел бы работать. (Тот проект тоже оказался странным – может быть, в этом есть закономерность?) В остальных случаях я тщательно разбивал работу на этапы, составлял график с критическим путем, обозначал риски, оставлял резервы и делал прочие правильные вещи. Большинство этих проектов были успешными: нужный продукт изготавливался в установленный срок в рамках выделенного бюджета. Однако немногие из этих проектов доставляли удовольствие.

Вот мои предложения по поводу создания странных проектов и, возможно, возведения их в норму вместо отклонения. Этот прием называется «пошаговое согласие» (incremental consensus); о нем рассказал мне в частной беседе консультант и писатель Джерри Вейнберг (Jerry Weinberg) [Weinberg] (я описал эту технику в [Phillips 2004]).

Пошаговое согласие начинается с того, что есть два человека, которые оба хотят работать в приятной обстановке над успешным проектом. Команда увеличивается путем добавления в нее по одному участнику. Когда один из участников команды находит человека, желающего в нее вступить (особое внимание уделяется его личному желанию), все остальные члены команды должны подтвердить свое согласие на присоединение этого нового человека. Никто не может присоединиться к команде без единодушного согласия ее нынешних участников. Скорее всего, в результате получится «странная» команда, в которой каждый из участников будет принят всеми остальными.

Команда моего странного проекта формировалась не по принципу пошагового согласия, потому что в ее составе около 100 человек. Однако десять руководящих сотрудников добавлялись в команду по одному, при этом каждый выражал желание участвовать в существующей команде (тот же принцип личного желания).

Возможно, мой странный проект не станет успешным. Мы столкнулись с несколькими техническими проблемами, которые грозят подорвать наш бюджет и затянуть поставку. Гарантировать успех проекта невозможно,

особенно если вы изобретаете устройства, придумываете алгоритмы и пытаетесь писать новые программы для работы на оборудовании заказчика. Но необычно, что мы открыто обсуждаем свои технические проблемы и ищем их решение как команда, озабоченная успехом проекта и благополучием каждого участника. Наличие второго обусловливает первое.

Ссылки

WEINBERG – Jerry Weinberg, *http://www.geraldmweinberg.com*

PHILLIPS 2004 – «The Software Project Manager's Handbook: Principles that Work at Work», IEEE Computer Society and Wiley-Interscience, 2004; Dwayne Phillips.

9.4. Помогите найти!

Забавность, интерес, удовольствие. Ведь это главное в программировании, правда? Те, для кого программирование – хлеб, вода и воздух, занимаются им, потому что оно доставляет им удовольствие.

Безусловно. Мы, правда, занимаемся этим также ради денег, положения, успеха и прочих обыденных вещей, о которых пишут книги. Но в основном мы занимаемся этим ради удовольствия. Вот почему никто еще не написал книгу «Программист на час» – не потому, что это более долгое занятие, а потому что никто не бросит его так скоро.

Однако после всего вышесказанного я должен сделать признание. Программирование *было* для меня удовольствием. Тридцать лет спустя я уже не испытываю прежних чувств. И переживая уже третий на моем веку кризис среднего возраста, пытаюсь выяснить причины этого. Пишу об этом в надежде, что кто-либо из читателей сможет мне помочь.

Вот что удалось выяснить мне самому. У меня есть ряд возможных объяснений, применимость которые к себе я пытаюсь проверить.

1. *Работодатели слишком ценят людей с 30-летним опытом, чтобы позволять им программировать.* Я просто не могу вспомнить, когда мне в последний раз по-настоящему предлагали программировать за деньги. Конечно, мне приходится изучать требования, составлять планы, проводить исследования, писать статьи. Но это другое. Удовольствие от программирования заключается в самом программировании. Ничем другим его не заменишь. А работодатели уверены, что молоденький программист ничуть не хуже старого и опытного.
2. *Программисты «выгорают».* То, в чем состоит удовольствие от программирования, составляет и его кошмар. Из тысяч мельчайших замысловатых деталей нужно соткать единый ковер, готовое произведение искусства. Со временем возня с этими мелкими деталями доставля-

ет все меньше удовольствия, превращаясь в рутину. По каким еще причинам столь многие программисты мечтают стать руководителями – какая от этого радость?

3. *Чем человек старше, тем он угрюмее.* Вот *это* действительно отвратительная мысль! Разве мы, многоопытные, так мало поняли в жизни, что с возрастом жить становится *тяжелее*, а не *легче*? С возрастом должна развиваться и способность получать удовольствие. Глядя на своих сверстников, я не вижу, чтобы это происходило.

Этот список можно было бы продолжить. И в этом я полагаюсь на вас, дорогой читатель. Но под конец рассказа хочу объяснить, как эта тема вообще возникла.

Не знаю, как это лучше выразить. В моем роду есть программисты следующего поколения. Лет тридцать назад у меня появился сын и, черт возьми, он решил пройти по отцовским стопам через всякую дрянь вроде программных багов и тому подобного. Итак, есть программист нового поколения, носящий мое имя. И, наблюдая его профессиональную карьеру, я вижу, как он получает удовольствие. Ностальгия и déjà vu... все в точности, как было у меня.

Тронула рассказанная им история с кольцом защиты записи. Как-то в пятницу вечером сидит он на работе, в программистской фирме, расслабившись после недели напряженного труда. И черт дернул одного из коллег швырнуть в другого кольцо защиты записи (не знаете, что такое кольцо защиты записи? Это пластиковая штуковина с дурацким названием, в отсутствие которой лента защищена от записи, а чтобы разрешить запись, нужно надеть на бобину с лентой это кольцо.) И, как и должно быть среди живых ребят, мяч был отбит. Кольца защиты записи летали по воздуху, как баги в неотлаженной программе. Даже вице-президент фирмы включился в игру (вы знаете, какие маленькие бывают теперь компьютерные фирмы и как молоды их руководители!). И вот мой сын, с широкой улыбкой на лице, с удовольствием описывал этот случай. А я, его слушатель, анализировал свою реакцию, в конце концов осознав, что это *зависть*. У меня в памяти мелькали картины счастливых дней на заре моей собственной жизни, когда мы на работе травили анекдоты, стреляли аптекарскими резинками и играли блицы в шахматы, как это положено. Что же со мной случилось?

В самом деле, что произошло? Я стал опытнее. Я стал больше зарабатывать, чувствовать большую ответственность. Я написал несколько книг, вел пару колонок в прессе. Занимался исследованиями, написал несколько статей. Я достиг успеха, о котором положено мечтать всем настоящим американским мальчикам и девочкам.

Но по дороге я что-то потерял. Может быть, вы подскажете мне, что?

Часть II

Как стимулировать творчество

Введение
10. Творчество в программирующей организации
11. Творчество в программных технологиях
12. Исторические даты творчества в программировании

Введение

В творческом процессе и при развитии новых идей есть две основные фазы: воображение и практическая работа. В фазе воображения вы генерируете идеи и играете с ними. В практической фазе вы проверяете их и реализуете.

В фазе воображения вы задаете вопросы: «а что, если?..», «а почему бы не...?», «какие законы можно нарушить?», «какие допущения можно отбросить?», «можно ли воспользоваться метафорой из другой области?». Девиз фазы воображения: «Нужно придумать что-то оригинальное».

В практической фазе вы задаете вопросы: «какую пользу может принести эта идея?», «хватит ли ресурсов для ее реализации?», «правильно ли определены сроки?», «кто может нам помочь?», «каков окончательный срок?», «что будет, если мы не достигнем цели?». Девиз практической фазы: «Нужно что-то сделать.

Роджер фон Эйк

«A Whack on the Side of the Head»,
Warner Books, 1990

Одно дело верить, что в работе над программным продуктом важно творчество, другое – стимулировать это творчество. Если предыдущая часть книги была посвящена философии и мотивам, то эта часть практическая.

И в популярной прессе, и в более серьезных журналах много говорится о том, как стимулировать творчество, но в компьютерной литературе, как мы увидим, об этом сказано мало. Такое впечатление, что увлечение дисциплиной, формальными и количественными методами разработки программного обеспечения, обсуждавшимися в части I книги, заставило позабыть о творческих методах.

К счастью, есть теоретик, который заметил эту проблему и собрался ее разрешить. В этой части, разделенной на несколько глав, представлены некоторые результаты работы покойного профессора Дж. Дэниела Кугера (J. Daniel Couger) из Университета Колорадо в Колорадо-Спрингс. Интерес Кугера к творчеству в программировании привел к появлению как ряда значительных исследований в этой области, так и Центра исследований творчества (Center for Research in Creativity, CRC) в этом учебном заведении. Весьма рекомендую приобрести и прочесть книги, написанные профессором Кугером по этому предмету.

Но довольно восторгов и рекламы! Вернемся к практике. Допустим, все прочитанное в этой книге убедило вас, что творчество – хотя бы отчасти – является важной частью процесса создания программного обеспечения. Допустим также, что вы руководите группой разработчиков программного обеспечения и горите желанием пробудить их творческие способности. Что для этого нужно?

Вот тут и пригодятся материалы профессора Кугера. В следующих главах, особенно в главе 11, отмечены его представления о творчестве при реализации информационных систем и приведены его рассуждения о том, где, когда и почему творчество может и должно иметь большое значение. Его конкретный пример, приведенный в конце главы 11, представляет огромный вклад в дело стимуляции творчества. Описан ряд приемов стимуляции творчества, изложена программа обучения этим приемам и рассмотрены результаты применения этой практики на конкретном предприятии. Мы привыкли считать, что инновации в предприятии внредряются сверху вниз, но тут профессор Кугер рассказывает о том, как они внедряются снизу вверх и даже доказывает, что такой способ может быть эффективнее!

Есть и ряд других важных источников предлагаемого материала. Вероятно, удивительно, что Уоттс Хамфри (Watts Humphrey), известный своими исследованиями процесса программирования, высказал довольно важные соображения об «управлении и новаторстве», приведенные в главе 10. Эти мысли Хамфри существенно дополняют труды Кугера. От Хамфри мы узнаем, что для управления новаторством в программировании руководителям требуется новое мышление, и выясняем, в чем оно состоит. Если это краткое изложение ранней книги Хамфри разожжет ваш аппетит, весьма рекомендую также «Managing for Innovation» (Управление и новаторство) или «Managing Technical People» (Как управлять техническими специалистами) из той же серии.

Ссылки

Некоторые интересные публикации о творчестве в программировании, развивающие представленный здесь материал:

- «Creative License», Better Software, March 2005; Patrick M. Bailey.
- «Imaginate», IEEE Software, Sept., 2004; Andy Hunt and Dave Thomas.
- «Provoking Creativity: Imagine What Your Requirements Could Be Like», IEEE Software, Sept. 2004; Neil Maiden, Alexis Gizikin, and Suzanne Robertson.
- «The Creative Edge» (a roundtable discussion), Atlanta Business Chronicle, Sept. 16, 2005; Powell Goldstein.
- «The Creativity Issue», Fast Company magazine, Dec., 2004.

Глава 10 | Творчество в программирующей организации

Введение
10.1. Греция и Рим: две очень разные культуры программирования
10.2. Контроль и корпоративная культура
10.3. Управление и новаторство
10.4. Творчество и стратегические информационные системы
10.5. Творчество против закона

Введение

Когда то, что требуется сделать, превысит ваши личные возможности, вам придется позаботиться о менеджменте. И чем крупнее проблема, тем сильнее придется углубиться в менеджмент.

Ричард Хемминг

«You and Your Research»,
(запись выступления на семинаре Bell Communication Research Colloquium Seminar 7 марта 1986 г.)

В этой главе представлено удивительное разнообразие точек зрения, и все они связаны с одной темой: как развить в программирующей организации дух творчества.

Представлены взгляды двух пар авторов, Питер Дегрейс/Лесли Сталь (Peter DeGrace/Leslie Stahl) и Стэн Рифкин/Чарльз Кокс (Stan Rifkin/Charles Cox), на огромные различия между теми организациями, где царит свобода, и теми, где установлена жесткая дисциплина. (Наверняка вы уже предположили, что скажут эти авторы, исходя из определений «свободы» и «жесткой дисциплины». Возможно, после прочтения их текстов ваше мнение изменится!) Мы последуем туда, куда нас приведет материал Уоттса Хамфри из «Управления новаторством», о котором говорилось во введении в часть II. Приводится также анализ стратегических информационных систем, основанный на вызвавшей противоречивые мнения статье Николаса Карра (Nicholas Carr) «IT Doesn't Matter» (ИТ ничего не дают), который может вызвать у вас сомнения относительно того, что в действительности пытался сказать Карр, когда рассердил стольких специалистов по ИС.

10.1. Греция и Рим: две очень разные культуры программирования

Я был рад прочитать то, что написал Питер Дегрейс (и его соавтор Лесли Сталь). Он не только выразил мои мысли. И не только сделал это удачно. Он еще и внес в свое изложение известную культурную широту. Посмотрите, к примеру, как он противопоставляет Грецию и Рим [DeGrace 1993].

Деграс описывает условия работы и культуру труда греков и римлян, отмечая большие и, как мне кажется, важные различия между ними. Греки, по его словам, работали как кустари или самостоятельные члены бригад. У них был личный инструмент, который они брали с собой на работу, а потом возвращались с ним домой. Они работали как отдельные независимые подрядчики, предоставляя свои услуги за плату. Они были вполне независимы.

Это сильно отличается от римских методов. В Риме определяющее значение имела принадлежность работника к общественной группе. Он приносил себя в жертву организации, отказываясь от своей индивидуальности и сливаясь с группой. По римским порядкам, работник приходил на рабочее место, брал общие инструменты и делал то, что ему сказано. Он не был самостоятельной личностью – душой и телом он принадлежал той организации, в которой работал. Однако за это он получал значительное вознаграждение. Оно имело вид защищенности, денег или власти.

Далее, Дегрейс заявляет, что описывает римский и греческий подходы без всякой моральной оценки. Не отдавая предпочтение ни тому, ни другому, он просто отмечает, что в условиях, сложившихся в нашей области, видит дисбаланс, поскольку большинство гуру предпочитает римскую, а не греческую точку зрения.

Но ирония здесь все же есть. Посмотрите, кому принадлежали согласно этим описаниям душа и тело отдельного работника. Боюсь, никто из нас не захотел бы, чтобы его душа и тело принадлежали компании, в которой он работает. Несколько лет я работал в той же аэрокосмической фирме, что и Дегрейс, когда он писал свою работу. Определенно это была компания римского типа, и если мы, ее работники, не испытывали ощущения, что наши тела и души ей принадлежат, то все же не сомневались, что компания ценит не личный вклад каждого, а работу группы. (В действительности, даже писать на компьютерные темы я начал в поисках выхода из этой безумно душной атмосферы! Из личных бесед с Дегрейсом я знаю, что он ощущал примерно то же.)

Есть, конечно, вещи и похуже организации греческого или римского типа. Некоторые менее цивилизованные фирмы, стоящие на низших ступенях организационного развития, вполне можно назвать «варварскими».

Давайте отобразим греко-римскую дихотомию в категориях, характерных для сферы разработки программного обеспечения. Для смеха добавим и варварскую систему (спасибо Стиву Макконнеллу за то, что вспомнил о ней). Читая следующий список, вспоминайте организации, в которых вам приходилось работать.

- Греки организуют вещи, римляне организуют людей, варвары почти ничего не организуют.
- Греческие методологии неформальны, римские – формальны, у варваров их нет.
- Греки пишут программы, римляне управляют проектами, варвары спешат кодировать.
- Греков вдохновляет задача, римлян – групповые цели, варваров – героика.
- Греки минимизируют объем создаваемой документации, римляне максимизируют его, варвары презирают любую документацию.
- Греки работают маленькими группами, римляне – крупными организациями, варвары – в одиночку.
- Греки используют в качестве инструмента вещи, римляне используют людей как инструмент, варвары не пользуются инструментом.
- Греки – демократы, римляне – империалисты, варвары – анархисты.
- Греки эмпирически индуктивны, римляне аналитически дедуктивны, варвары немыслимо эмоциональны.
- Греки интуитивны, римляне логичны, варвары импульсивны.
- У греков управление основано на заслугах, у римлян – на функциях, у варваров – на страхе.
- Грекам важна суть, римлянам – форма, варварам – количество строчек кода.
- Греки делают дело, римляне планируют дело, варвары портят дело.

В некотором смысле эти различия предвосхитили дебаты по поводу Agile-программирования и процесса, начавшиеся почти на десять лет позже. (Эта тема уже обсуждалась в нашей книге.) Греки хорошо вписались бы в лагерь Agile, римляне усиленно трудились бы над повышением своего уровня CMM, а варвары с негодованием отвергли бы и то и другое!

Под конец [DeGrace 1993] Дегрейс приводит замечательную и ужасную историю, свидетелем которой был не он один – в разные времена и в раз-

ных местах она случалась и с другими. Это рассказ о том, как жили-были два программиста, которым дали решать одну и ту же задачу.

Первый программист увидел, что задачу можно решить традиционными методами. Он собрал команду, чтобы разработать решение. При этом он развил заметную деятельность: совещания, планы, документы, отчеты и так далее.

Второй программист увидел в задаче возможность проявить свое умение решать задачи и занялся обдумыванием решения, не демонстрируя заметной активности.

Первый программист предложил детальное решение со столь же детальными графиками отладки и тестирования. Второй программист предложил простое решение, не связанное с какими-либо проблемами.

Чем все кончилось? Первому программисту – почет и уважение, поскольку (к примеру) у него оказалась выше производительность, измеренная в строках кода. А второму – жесткая критика. Пусть его решение безупречно, но где же горы планов и документации, где труд в поте лица?

Очевидно, в этой притче о греках и римлянах справедливость не восторжествовала. В данном случае, я бы сказал, состоялось «награждение виновных и наказание невинных». Такое положение дел не удивительно (хотя, несомненно, прискорбно), ибо сегодня авторитеты в мире программирования отдают предпочтение римской, а не греческой точке зрения.

Можно пофантазировать, что будет, если в эту историю ввести еще и программиста-варвара. Например, он может с безумной скоростью писать код, полный ошибок. Тем не менее, он может завершить работу раньше всех, если остальные старались написать корректно работающую программу, и в результате сорвать поначалу аплодисменты. Зато когда начнут поступать сообщения об ошибках, программисту-варвару придется заниматься отладкой, забыв об отдыхе и все чаще оказываясь на критическом пути, и потом выступить этаким героем, спасшим горящий проект (хотя пожар он сам же и устроил). В этом случае самому отъявленному преступнику будут возданы самые высокие почести!

Конечно, этот сравнительный анализ Рима и Греции с примесью варварства нельзя назвать историей со счастливым концом. Но он прекрасно отражает некоторые проблемы, рассматриваемые далее в этой книге.

Ссылки

DeGrace 1993 – The Olduvai Imperative, Prentice-Hall, 1993; Peter DeGrace and Leslie Hulet Stahl.

10.2. Контроль и корпоративная культура

В корпоративной культуре разных компаний преобладает один из двух следующих элементов:

- Руководители, задача которых – контролировать, чтобы добиться превосходства.
- Технические специалисты, задача которых – экспериментировать, чтобы добиться превосходства.

Разумеется, это отголоски борьбы римского и греческого начал, которую мы только что видели. Точка зрения, которую предлагают Питер Дегрейс и Лесли Сталь, ясна: есть греческие организации, где вернейшим путем к хорошему продукту считаются умения и таланты технических специалистов, и есть римские организации, где вернейшим путем к хорошему продукту считаются оркестровка и контроль со стороны руководства.

Этот *олдувайский императив* (The Olduvai Imperative) – единственное место, где наблюдается подобная дихотомия взглядов? Оказывается, нет. Независимо и в совершенно другой части программного леса к тому же выводу примерно в то же время пришли Стэн Рифкин (Stan Rifkin), работавший тогда в Институте программирования (SEI), и Чарльз Кокс (Charles Cox) [Rifkin 1991].

Если Дегрейс и Сталь формировали свои взгляды в аэрокосмической компании Boeing, участвуя в различных программных проектах, то Рифкин и Кокс пришли к аналогичным культурологическим выводам, занимаясь исследованиями при поддержке SEI.

Они исследовали, почему в одних организациях придумывают и применяют программные метрики лучше, чем в других. В ходе исследования они посетили одно или несколько подразделений следующих компаний: Contel, Digital Equipment, Hewlett-Packard, Hughes Aircraft, IBM, McDonnell Douglas, NASA, NCR и TRW. Что же они там выяснили?

То, что они выяснили, сказано в самом начале этого раздела. Именно Рифкин и Кокс пришли к выводу, что в одних компаниях главенствуют руководители/контроль, а в других – технические специалисты/эксперименты, о чем они публично и независимо объявили примерно в то время, когда Дегрейс и Сталь наводили последний лоск на свою книгу.

Но Рифкин и Кокс дополнили их квалифицированное и обоснованное мнение одним новым измерением. В своей работе Рифкин и Кокс обнаружили, что компании, отдающие преимущество технической стороне и экспериментам, лидируют в успешном применении метрик. А что другие компании, те, что отдают предпочтение руководству и контролю? Они применяли метрики не так успешно.

В качестве примеров ориентированных на технологии компаний Рифкин привел Digital Equipment и Hewlett-Packard. Большинство компаний, по его словам, относится к другому типу. Для них характерны иерархическая система организации и повышенное внимание управлению.

Как организовано дело в компаниях, успешно применяющих метрики?

- Обычно они выходят на уровни 4 и 5 (очень хорошо) процесса SEI, минуя уровень 3. То есть они применяют количественные показатели и обратную связь для совершенствования своего процесса, даже не имея определенного процесса! (Вот и конец дискуссии относительно руководителей/контроля и технических специалистов/экспериментов.)
- Ошибки выведены за рамки «криминала». О том, что было сделано неправильно, говорится открыто, чтобы это не повторилось впредь. Нет нужды скрывать свой провал: руководству запрещено или оно просто не считает нужным наказывать за ошибки.
- Количественные показатели являются частью повседневной работы. Количественные оценки применяются не по особому распоряжению или в связи с политикой, а по общему согласию, что только так и можно строить продукт.

В таком случае Рифкин и Кокс не только поддерживают точку зрения Дегрейса/Сталь на греко-романские культуры в программировании, но и определенно свидетельствуют (конечно, довольно ограниченно) в пользу того, что греческий путь (в терминологии Рифкина/Кокса это технические специалисты/эксперименты) обладает некоторыми преимуществами.

Ни Рифкин/Кокс, ни Дегрейс/Сталь в своих изысканиях не придают особого значения слову «творчество». Однако и не обладая богатым воображением можно связать метод технических специалистов/экспериментов с большими возможностями для творчества, чем метод руководителей/контроля. Возможно, лучше всего это удалось Майклу Кузумано (Michael A. Cusumano) в книге «Japan's Software Factories» (Фабричное производство программ в Японии), когда, обсуждая выгоды и проблемы промышленного производства программ, он пишет [Cusumano 1991]:

> «С одной стороны, чрезмерное увлечение индивидуальным творчеством и независимостью может создать проблемы для управления стоимостью разработки, ее качеством и сопровождением в течение длительного периода. С другой стороны, избыток организованности и контроля может удушить творчество, новаторство и способность к изменению – как и вызвать протест со стороны менеджеров, разработчиков, а, возможно, и клиентов».

Интуитивно мы, конечно, склонны согласиться с Кузумано в том, что творчество и контроль в какой-то мере враждебны друг другу. А это подкрепляет мысль о том, что путь технических специалистов/экспериментов теснее связан с творчеством в программировании, чем путь руководителей/контроля.

Ссылки

RIFKIN 1991 – «Measurement in Practice», CMU/SEI-91-TR-16, also published in *Proceedings of the Sixteenth Annual Software Engineering Workshop*, NASA Goddard Space Flight Center, December 1991; Stan Rifkin and Charles Cox.

CUSUMANO 1991 – «Japan's Software Factories», Oxford University Press, 1991; Michael A. Cusumano.

10.3. Управление и новаторство

Уоттс Хамфри (Watts Humphrey) более всего знаменит своей работой над процессом разработки ПО в SEI. Его работа «Managing the Software Process» (Управление процессом разработки ПО) [Humphreys 1989] и другие на ту же тему, пожалуй, одни из наиболее важных в области программного инжиниринга, а сама эта книга вызвала самые важные дискуссии по программному инжинирингу за последние два десятилетия.

За два года до первой книги, посвященной процессу, Уоттс написал другую книгу, «Managing for Innovation» (Управление и новаторство) [Humphreys 1987]. Эта более ранняя книга годами не привлекала почти никакого внимания, хотя во многом была не менее важна, чем та, которая хорошо известна. Спустя восемь лет она была переиздана под названием «Managing Technical People» (Как управлять техническими специалистами) [Humphreys 1997], и, насколько мне известно, версия 2.0 этой книги прошла такой же незамеченной, как и предыдущая.

Подзаголовок «Leading Technical People» (Руководство техническими специалистами) книги о новаторстве перешел в название ее следующей версии, а суть обеих книг в том, что при создании программного продукта роль человека очень важна. Книги посвящены программистам-новаторам и роли руководства в поддержке новаторства. Эти книги удивительным образом соседствуют с последующими работами Хамфри, посвященными процессу, потому что позже он занял позицию, согласно которой ключ к успешной программе – процесс (а не люди).

Но вернемся к более ранней книге Хамфри и ее последующей версии 2.0: какова связь между новаторством и творчеством? Хамфри, цитируя Теодора Левитта, описывает ее очень кратко:

> «Творчество – это придумывание нового. Новаторство – это создание нового».
>
> «Творчество необходимо для новаторства, но для него также требуется большой нелегкий труд», – добавляет Хамфри.

Книга посвящена двум темам – творчеству и новаторству, хотя Хамфри явно больше заинтересован в последнем, поскольку именно в «создании» и «нелегком труде» роль управления выдвигается на передний план.

Что мы находим у Хамфри по поводу запутанной связи между управлением и творческим техническим работником?

Относительно контроля он приводит сведения из литературы, согласно которым «руководители, жестко контролирующие своих подчиненных, обычно получают существенно меньше творчества, чем те, кто управляет в более свободном и неформальном стиле».

Относительно технической подготовки руководителей он приводит другие сведения, согласно которым «если знания руководителя невелики, уровень новаторства тем выше, чем большая свобода предоставлена членам группы».

«Если руководитель имеет высокую квалификацию в администрировании, управлении персоналом и технологиях, – пишет Хамфри дальше, – то результаты могут варьировать: иногда свобода идет на пользу, иногда – нет». В заключение Хамфри приводит старую поговорку: «Если не разбираешься в деле, лучше не мешай».

Взглянув на эту проблему с другой точки зрения, Хамфри рассматривает результаты изучения успешных и не успешных групп. «В самых новаторских из них руководители лично участвовали в работе, поддерживая тесный технический контакт со своими работниками. В наименее новаторских руководители были не столь активны и более далеки от работников».

Какой урок можно извлечь из этого материала? Возможно, тот, что «наивысший уровень новаторства достигается, если а) руководитель обладает техническими знаниями и участвует в работе группы, при этом б) предоставляя своим техническим специалистам наивысший уровень свободы».

Является ли техническое творчество важнейшим фактором успешности новаторства? Согласно Хамфри и приводимым им результатам других исследований, нет. В действительности, «постоянная склонность технических специалистов ограничивать себя стенами лаборатории, не занимаясь подробным изучением запросов пользователей» является, по его словам, одной из главных причин неудач научно-исследовательских работ.

Есть ли альтернатива у технического специалиста, замкнутого в стенах лаборатории? «Разработка успешного проекта в большей мере зависит от тонкого понимания рынка, чем от технической компетентности», – говорит он, цитируя исследование, проведенное в Пенсильванском университете. Согласно другому исследованию новаторство наиболее успешно, «когда пользователи технически грамотны». Имеется в виду, что у пользователя, понимающего, какую задачу решает приложение, больше шансов предложить новаторское решение, чем у компьютерного специалиста, который разбирается только в том, какую вычислительную задачу нужно решить. И Хамфри цитирует еще одно исследование, выявившее, что «75% всех инноваций... появляются благодаря рынку, в то время как наибольший процент технических инноваций ни в одном исследовании не превысил 34%». Таким образом, успешность новаторства определяют потребности клиентов, а не технологический прогресс. (Что-то знакомое, правда? Вариации на старую тему «нужда заставит».)

Интересное отступление в книге касается проблемы «влияния возраста на творчество». Распространено мнение, что творческие и новаторские способности достигают своего пика в достаточно раннем возрасте (по приводимым данным, от 21 до 29 лет), но Хамфри заявляет, что «количество опровергающих это свидетельств растет». В одном исследовании было обнаружено, что «результативность достигала максимума в весьма раннем возрасте, но затем снижалась весьма незначительно». «Момент достижения максимума зависит от области знаний: он наступает раньше в абстрактных областях, вроде математики или теоретической физики, и позднее в таких практических специальностях, как биология или геология. Начальный пик приходится на середину третьего десятилетия, но был обнаружен и поздний пик в середине и конце пятого десятилетия. Эти два пика были обнаружены во всех обследованных группах инженеров и ученых, и провал между ними не имел большого значения».

Хамфри подводит итог ободряющей (по крайней мере, для читателей в возрасте!) мыслью: «Для творческих людей возраст на границе 30–40 лет сопряжен с большими стрессами. Однако преодолев этот барьер, многие инженеры и ученые продолжают творчески работать в течение долгих лет... многие из 1100 изобретений Томаса Эдисона появились ближе к концу его жизни, продолжавшейся 84 года».

В таком случае, с точки зрения «управления и новаторства», порядочный руководитель должен учитывать, что все его сотрудники независимо от их возраста могут внести творческий и новаторский вклад.

В книге Хамфри о новаторстве есть какой-то странный диссонанс. До того как поступить в SEI, Хамфри долгие годы проработал в IBM. Многие из приведенных Хамфри историй и значительная часть раздела благодарно-

стей (особенно в первом издании книги) связаны с его работой в IBM. Сейчас, когда уже все знают о тогдашних трудностях IBM, когда любой колумнист компьютерного издания волен разглагольствовать на тему «что случилось с IBM», несколько странно читать, что «IBM – прекрасный пример технологического успеха корпорации. Она продемонстрировала удивительное умение принимать правильные решения и правильно принимать решения». Лучше не обращать на это внимание в первой книге (в версии 2.0 Хамфри благоразумно умерил свои похвалы в адрес IBM), поскольку то, что Хамфри говорит о новаторстве, творчестве и управлении техническими специалистами, безусловно заслуживает прочтения.

Ссылки

HUMPHREY 1997 – «Managing Technical People», Addison-Wesley, 1997; Watts S. Humphrey.

HUMPHREY 1989 – «Managing the Software Process», Addison-Wesley, 1989; Watts S. Humphrey.

HUMPHREY 1987 – «Managing for Innovation», Prentice-Hall, 1987; Watts S. Humphrey.

10.4. Творчество и стратегические информационные системы

О стратегических информационных системах (Strategic Information Systems, SIS) много спорят. Те, кто верит в их значение (а таких немало), применяют к ним некий псевдоним «Competitive Advantage Systems» (Системы обеспечения конкурентного преимущества), подразумевая, что удачная стратегическая система принесет конкурентные преимущества применяющей ее организации. А разве преимущество перед конкурентами – не наибольшая польза, которую информационные технологии могут принести своему предприятию?

Но есть и критики SIS. Наиболее заметным из них стал, видимо, Николас Карр (Nicholas Carr), вызвавший крупный переполох в области информационных технологий, если не вообще в бизнесе, своим знаменитым очерком «IT Doesn't Matter» (ИТ ничего не дают) [Carr 2003]. Суть его выступления, скрытая за провокационным названием, заключалась в том, что успех стратегических информационных систем оказывается все более сомнительным. И дело было не в том, что невежественные массы бросились опровергать статью из-за ее заголовка, но именно в SIS.

Почему Карр взялся за SIS? Потому, что если взглянуть на список успешных SIS, который есть почти в любом учебнике начального уровня по ИТ,

то обнаружится, что большинство из них создано несколько десятков лет назад. Успешные SIS были, как вынужден признать Карр и всякий, кто объективно смотрит на эту область, но в последнее время их гораздо меньше. В том списке есть такие золотые шедевры прошлого, как система заказа авиабилетов (передовая для своего времени работа American Airlines более чем сорокалетней давности) и система складирования/доставки (Wal-Mart применяет ее несколько десятилетий). Любое предприятие с большой радостью завело бы себе SIS, говорят эти скептики, но вероятность получить ее в наши дни весьма мала. Вот почему «ИТ ничего не дают»: дело не в том, что ИТ бесполезны для предприятия, а в том, что они редко помогают ему в стратегиях и повышении конкурентоспособности.

(Любопытная аналогия прослеживается между статьями «ИТ ничего не дают» Карра и знаменитой «Серебряной пули нет» Фреда Брукса. В своей статье Брукс высказал мнение, что настоящие серебряные пули (революционные прорывы вперед) в программном инжиниринге были и остались в далеком прошлом и руководителям не стоит рассчитывать на появление чего-то нового, революционного в смысле возможностей создавать программные системы. То же самое Карр говорит о SIS/системах достижения конкурентного преимущества.)

Но ввиду важности SIS для предприятий, отсутствие новых положительных результатов не должно вести к отказу от их поиска. Нужно считаться с реальностью: попытки достичь конкурентного преимущества с помощью информационных технологий скорее всего обречены на провал. Но это ни в коем случае не должно мешать предприятию действовать в этом направлении.

С учетом этих обстоятельств, как предприятию взяться за определение и строительство SIS? Обратимся за помощью к авторам, которые на протяжении ряда лет немало сказали на эту тему. Например, в двух научных статьях [Galletta, Sampler, and Teng 1992] и [Galletta and Sampler 1993] мы находим огромное количество мыслей о том, откуда берутся основополагающие идеи успешной SIS. И все эти мысли, по мнению авторов, сводятся к творческому подходу. Чтобы получить систему, по-настоящему обеспечивающую конкурентное преимущество, необходимо, говорят они, предельное напряжение творческой мысли.

Однако – и в этом заключается ирония – Галетта и его коллеги отмечают, что немногие из авторов, изучавших SIS, продемонстрировали понимание этого. В литературе по SIS, которую они старательно изучали, почти ничего не говорится о связи SIS и творчества. Возможно, отсутствие понимания роли творчества и привело к тому, что первоначальный расчет на широкое распространение SIS среди предприятий оказался глубоко ошибочным. Ведь если бы для определения и реализации SIS не требова-

лось творчество, они были бы на каждом предприятии! А если оно необходимо, то неудивительно, что этих систем так мало.

Затем в этих статьях обращается внимание на то, как стимулировать творческое мышление, способное привести к успешным SIS.

В результате обзора литературы, посвященной SIS, было выделено несколько схем, применяемых практиками и учеными в поиске идей для SIS. Вот эти методы:

1. Анализ факторов, влияющих на конкурентоспособность. К ним относятся: а) создание препятствий для появления конкурентов, б) увеличение цены, уплачиваемой клиентом в случае перехода к конкуренту, в) усиление позиций в отношениях с поставщиками, г) поиск возможностей существенного сокращения издержек и д) предложение новых продуктов или услуг.
2. Изучение цепочки добавления стоимости – от получения сырья до предоставления продукта или услуги.
3. Изучение жизненного цикла клиентских ресурсов и выявление процессов, посредством которых клиенты взаимодействуют с предприятием.
4. Выработка вариантов стратегических решений путем ответа на вопросы: а) Какова наша стратегическая цель? б) Какой стратегический удар мы можем нанести по этой цели? в) Каков стратегический характер возможных действий (наступательный, оборонительный)? г) В каком направлении должен быть нанесен этот удар? д) Какие возможности информационных систем могут быть использованы?

Но Галетта с коллегами также скептически замечают, что не все согласны с действенностью этих методов. Они приводят мнение одного автора о том, что «стратегические системы проходят длительный процесс адаптации, а не совершают крупный прорыв, быстро приносящий успех... нельзя просто взять и написать SIS».

Несмотря на это, авторы все же настаивают, что можно повернуть свое предприятие в сторону создания собственной SIS, и успех этого начинания зависит от усилий по стимуляции творчества. Затем они указывают три социологических уровня, где может происходить такая стимуляция творчества, и способы ее осуществления.

1. Творческие люди. Галетта с коллегами отмечают, что исследователи проделали большую работу в поисках методов выявления творческих личностей. У творческих людей может просто быть а) врожденная творческая жилка (хотя обнаружить таких людей и непросто!), у творческих людей может быть значительное б) фактическое знание дела (это условие необходимое, но ни в коей мере не достаточное), и творче-

ские люди должны в) обладать мотивацией (чтобы они могли реализовать свои предложения) (и опять, это условие необходимое, но ни в коей мере не достаточное).

2. Творческие группы. Большая часть проделанной научной работы посвящена стимуляции творчества в группах. Некоторые полученные результаты описаны в этой книге далее. Здесь же отметим лишь, что в число предлагаемых методов входят «исследование граничных условий» (для проверки сделанных предположений), «принятие желаемого за действительное» (когда допускается предлагать «невозможные» решения), «ориентация на цель» (когда допускается облегченное восприятие проблемы без учета возможных препятствий и ограничений), «мозговая атака» и «обратная мозговая атака» (выдвижение и оценка самых разнообразных решений), «нелогичные стимулы» (когда поощряются разные «сумасшедшие» методы) и «поименный групповой метод» (отдельные люди вносят свои предложения, а группа принимает решение).
3. Творческие организации. Пожалуй, эта тема наиболее досконально изучается в работе Галетты и его коллег, посвященной роли творчества для SIS. Они предложили много подходов:
   a) поощрение упомянутых выше схем SIS;
   b) привлечение творческих работников (с этим связана обсуждавшаяся выше проблема их выявления);
   c) привлечение знающих работников (хорошо знакомых с проблемной областью);
   d) привлечение людей с широким кругозором (способных замечать неожиданные связи между фактами);
   e) применение когнитивных средств (инструменты, заметки, картинки, диаграммы, аналогии);
   f) развитие внутренней мотивации (и отказ от внешних стимулов вроде премий или нереальных конечных сроков, которые, по общему мнению, отрицательно влияют на творчество);
   g) диверсификация состава групп (разнообразие уровня подготовки и опыта);
   h) применение творческих методов (о них будет сказано ниже);
   i) поддержка группы (установление реальных сроков, свобода дискуссий, приверженность цели...);
   j) уничтожение организационных барьеров (явная поддержка альтернативных взглядов);
   k) поощрение готовности к риску (неудача допускается как возможный вариант).

Решая проблему SIS на всех трех социологических уровнях, по мнению наших авторов, предприятие увеличивает свой шанс получить действительно полезную SIS.

Одна из связанных с SIS проблем – обеспечение долгосрочности обеспечиваемых ими преимуществ. Как только об успешной SIS становится известно в обществе, едва ли не все другие предприятия бросаются организовывать себе аналогичную с помощью своих ИТ-специалистов. Система продажи авиабилетов American Airline перестала быть конкурентным преимуществом, когда за последующее десятилетие все другие авиалинии завели у себя такие же системы. У Wal-Mart по-прежнему функционирует сложная система управления поставками и складированием, но среди крупных предприятий розничной торговли у каждого есть своя версия такой системы.

Итак, есть две категории/фазы разработки SIS: одни выдвигают оригинальную мысль и успешно ее реализуют; другие, заметив удачную идею, бросаются вдогонку. Интересно поразмыслить, чем первопроходцы отличаются от подражателей. Вполне можно предположить, что для первых огромную роль играет творчество, гораздо менее важное для вторых. Если это так, то описанные выше творческие подходы могут иметь существенное значение для первопроходцев и, видимо, меньшее для подражателей. (Отметим, что с учетом важности SIS для предприятия не следует порицать тех, кто просто подражает. Тот, кто сегодня выступает первопроходцем, завтра вполне может копировать оригинальные идеи своего конкурента.)

Ссылки

Brooks 1995 – «Серебряной пули нет» («No Silver Bullet»), первая публикация в 1986 году в Proceedings of the IFIP Tenth World Computing Conference; позднее – заглавная статья Communications of the ACM; впоследствии опубликовано в «The Mythical Man-Month» («Мифический человеко-месяц», юбилейное издание), Addison-Wesley, 1995; Frederick P. Brooks, Jr.

Carr 2003 – «IT Doesn't Matter», Harvard Business Review, May 2003; Nicholas G. Carr.

Galletta, Sampler, and Teng 1992 – «Strategies for Integrating Creativity Principles into the System Development Process», Proceedings of the 25th Annual Hawaii International Conference on Systems Sciences, Jan., 1992; D.F. Galletta, J. L. Sampler, and J. T. C. Teng.

Galletta and Sampler 1993 – «Creativity: The Missing Ingredient in Identifying Strategic Informaton Systems», working paper WP-703RR, University of Pittsburgh, Katz Graduate School of Business, Sept. 1993; D.F. Galletta and J. L.

10.5. Творчество против закона

Над вопросом о том, как стимулировать творчество в программировании, висит дамоклов меч. И мечом этим является, как ни странно, закон. Нравится это нам или нет, но законодательство имеет большое отношение к тому, как проявляет себя творчество.

Какие законы имеются в виду? Те, которые относятся к защите интеллектуальной собственности: патентное право, законы об авторских правах, о коммерческой тайне и даже о наемном труде.

В современном программировании эти законы очень активно обсждаются. Что делают законы об интеллектуальной собственности – стимулируют творчество или подавляют его?

Цель этого раздела – осветить данную проблему, не претендуя, конечно, на ее решение (что выходит за рамки возможностей автора и системы правосудия в целом! Споры по этому вопросу еще долго будут вестись в мире юриспруденции и программирования после того, как книга выйдет из печати, а потом исчезнет с книжных полок).

Вот в чем суть этих споров.

В одном углу ринга, в синих трусах Защитника социальных свобод личности, – убеждение, что законы об интеллектуальной собственности подавляют творчество, возводя стены вокруг новых идей и ограничивая право новаторов свободно распространять информацию о них.

В другом углу, в красных трусах Защитника свободы предпринимательства и права на получение прибыли для отдельных лиц и корпораций, – убеждение, что законы об интеллектуальной собственности способствуют творчеству, охраняя новатора от кражи его идей и прибыли.

Главное, в чем различаются эти точки зрения, – это вопрос мотивации. Правда ли, что новаторов побуждает к труду желание быть частью процесса интеллектуальной свободы, когда они стоят на крепких плечах других новаторов – их предшественников? Или новаторами движет стремление к получению прибыли, без которого они ни на что не способны? Несмотря на то что некоторые пытаются доказывать ту или другую точку зрения, на самом деле здесь нет правых и заблуждающихся. Есть просто две разные точки зрения.

Мы, программисты, обычно считаем, что этот диспут возник лишь в наше время и порожден быстрым развитием и распространением программного обеспечения как научной дисциплины, продукта и средства получения прибыли. На самом деле, все совершенно иначе.

Этот диспут стар, как и само новаторство. Например, в [Wagner 2004] мы видим, что те же споры активно велись еще 120 назад. Томас Эдисон, до

сих пор остающийся одним из крупнейших патентовладельцев в США, подал 1093 патентные заявки на свои различные изобретения. Его противники выдвигали разные возражения – от несостоятельности патентуемых им вещей до недостаточной квалификации Патентного бюро США, чтобы судить о работе Эдисона. Например, новомодная вещица, названная электрической лампой, вызвала, как говорится в статье, «бурю споров». Действительно ли это было оригинальным изобретением или Эдисон использовал работы других, а его собственный вклад был незначительным? Было ли оно настолько важным, что заслуживало патента? Было ли Патентное бюро достаточно компетентным для принятия решений по этим вопросам?

Конечно, патентное право – не единственный источник споров. Дискуссии по поводу авторского права и коммерческой тайны в принципе аналогичны. Движение Open Source (программирование с открытым исходным кодом) и близкие к нему, склоняющиеся на сторону Синих трусов Социальной свободы, придумали юридические схемы под названием «Creative Commons» и «CopyLeft», во многих отношениях противоположные охране авторских прав, поскольку их цель – свобода распространения программных продуктов, а не способы его ограничения. И, разумеется, когда исходный код становится «открытым», о какой-либо защите коммерческой тайны и речи идти не может.

В действительности существует история и ряд заметных тенденций в развитии этих проблем интеллектуальной собственности. Юрист Роланд Коулз (Roland Coles) рассматривает период до 1975 года как эру промышленных секретов, период 1975–1990 годов как эпоху авторских прав, а с 1990 года и до наших дней – как эпоху патентов [Cole 2005]. Этим объясняется, почему в последнее время такое внимание обращалось на патенты – их «за» и «против».

Во всей этой дискуссии есть интересная пограничная область. Стив Макконнелл, участвовавший в обсуждении данной книги, счел весь этот спор до некоторой степени бурей в стакане воды. Программирование и программные продукты меняются, по его мнению, так быстро, что практически невозможно определить, является ли некий конкретный метод новым/оригинальным/допускающим защиту или всего лишь версией того, что давно известно. («Есть столько разных способов сделать одно и то же, – говорит он, – что современные методы защиты оказываются практически бесполезными»). Любопытным примером здесь послужит описанный в [Wagner 2004] судебный иск к Microsoft, поданный компанией Eolas Technologies. Фирма заявила, что ей принадлежат патенты на некие расширения веб-страниц, которые Microsoft применила в своих продуктах, в частности в Internet Explorer. Однако Патентное бюро не согласилось с этим, указав, что запатентованная Eolas работа основана на более

ранней, выполненной Тимом Бернерсом-Ли за десять лет до того. (Заметим, что если патентное бюро допустило ошибку, выдав первоначальный патент, то его юридические процедуры позволили исправить ее, когда это потребовалось.)

Насколько важен данный спор? Критики нынешней юридической системы предвидят «смерть новаторства в программировании». Другие, напротив, ожидают множества инноваций, указывая на то, как вокруг новых программных продуктов возникают целые огромные отрасли (например, использующие Интернет), которых до недавнего времени не существовало.

Это вариант «твоя платит деньги и выбирает». Действительно ли это «творчество против закона», как сказано в заголовке раздела (в основном для возбуждения читательского любопытства)? Или это «творчество плюс закон», когда закон действительно стимулирует творчество путем его защиты? Выбор за вами. В любом случае никто не сможет вас опровергнуть!

Ссылки

Cole 2005 – Cited in «Open Source Myths and Realities Presented», The Software Practitioner, May 2005.

Wagner 2004 – «For Innovation's Sake», InternetNews.com, Nov. 9, 2004; Jim Wagner.

Глава 11 | Творчество в программных технологиях

Введение

11.1. Примеры творческого подхода к реализации информационных систем

11.2. Творчество и разработка ПО: недостающее звено

11.3. Творчество и реальность: конкретный случай

Введение

Возможности программ ограничены главным образом человеческим воображением. С учетом огромного творческого потенциала человека и продолжающегося стремительного развития аппаратного обеспечения компьютеров нелепо было бы полагать, что программные технологии достигли зрелости.

Майкл Кузумано

«The Business of Software»,
Free Press, 2004

Здесь результаты Дэна Кугера, о которых говорилось во введении в часть II, и покажут себя в полной мере. В трудах того, кто успешно соединил в своей профессиональной деятельности творчество и разработку информационных систем, мы находим исключительно ценный материал. Например, описываемые им конкретные случаи способны добавить вдохновения всем, кто подумывает о внедрении стимулирующих творчество методов в программирующей организации. Приводимые им примеры творческого подхода трудно игнорировать!

11.1. Примеры творческого подхода к реализации информационных систем

Большинство профессионалов в области компьютеров и программирования согласится с тем, что творчество составляет важную часть умения строить программные системы. Можно спорить о том, *где* именно требуется это творчество в процессе создания программных систем, и некоторые дискуссии по этому поводу приведены ранее в нашей книге, но с утверждением *в целом* трудно не согласиться.

Тогда почему тема творчества почти не затрагивается в компьютерной литературе? Покойный проф. Дэниел Кугер один из немногих исследовал роль творчества в отдельно интересующей его области информационных систем. Изучив то, что написано о творчестве, он выяснил следующее:

- опубликовано более 4000 статей о творчестве (напомню, что обзор проводился около 10 лет назад);
- лишь 5 статей из этого числа появилось за 35-летнюю историю литературы по информационным системам.

Профессор Кугер назвал это «загадкой». Похоже, профессиональные компьютерщики и программисты настолько внутренне замкнулись на технических проблемах создания успешных информационных систем, что у них не осталось сил на то, что находится вне этих узких границ.

Устранение этого недостатка стало одной из жизненных целей проф. Кугера. Он много и хорошо писал о роли творчества в информационных системах и внес большой вклад в эту скудно освещаемую область. Едва ли не самым значительным из написанного им стала книга «Creativity and Innovation in Information Systems Organizations» (Творчество и новаторство в организациях, использующих информационные системы) [Couger 1996]), которая уже больше десяти лет остается удивительно полной и современной. В данном разделе я ссылаюсь на его работу, посвященную роли творчества в жизненном цикле программы, чтобы рассмотреть конкретные средства стимулирования творчества у программистов [Couger 1990].

В академических трудах обсуждение относительно новой темы начинается с соответствующих определений, и профессор Кугер поддерживает эту традицию. Правда, определение главного понятия «творчество» далось ему не без труда, поскольку в литературе он обнаружил более 100 различных определений! В конце концов, он взял за основу «наиболее полное», по его мнению, определение, найденное у [Newell and Shaw 1962], которые написали, что творчество в решении должно удовлетворять одному или нескольким из следующих условий:

- результат обдумывания отличается новизной или ценностью (для того, кто думает, или для его области);
- обдумывание оказывается нетрадиционным, то есть требует изменения или отказа от ранее принятых концепций;
- обдумывание требует высокой мотивации и настойчивости, отличается большой продолжительностью (непрерывно или с промежутками) или высоким напряжением;
- начальная постановка задачи неясна или некорректна, поэтому формулирование самой проблемы входит в задачу.

Кугер суммировал все это, сказав, что в своей работе о творчестве решил охарактеризовать ее предмет как обладающий уникальностью и полезностью.

Теперь о жизненном цикле программы, каким он представляется Кугеру. В начале он усмотрел необходимость выделить в общем жизненном цикле время для «оценки творчества» – моменты, в которые разработчикам системы следует остановиться и изучить возможность применения более творческих методов, чем те, которые они первоначально избрали. Он считает, что такие моменты наступают в конце этапов жизненного цикла, которые он считает предварительными: *определение требований*, *логическое проектирование*, *физическое проектирование*, *программное проектирование* и *программирование*. Он считает, что такие стратегические паузы занимают очень незначительную долю всех ресурсов проекта – меньше полпроцента, исходя из его представлений о типовых проектах и деятельности, необходимой для оценки творчества.

Однако, отметил он, «мы, к сожалению, не можем получить фактические данные о применении творческих приемов в разработке ИС», поскольку... «автор не нашел организаций, в которых бы они применялись». (Ниже мы изучим приведенный проф. Кугером конкретный пример, где он попытался устранить этот недостаток, вводя творческие методы в реальном времени в реально действующей организации и оценивая результаты!)

Хотя в своей работе он разбил творческие методы по различным этапам жизненного цикла, к одному творческому приему он обращался на всех этапах – проф. Кугер назвал его «5W и H». Так он обозначил вопросы «кто» (Who), «какой» (What), «где» (Where), «когда» (When), «зачем» (Why) и «как» (How), которые нужно задавать, чтобы гарантированно изучить весь спектр вариантов задачи и ее решения (журналист узнает здесь список вопросов, которые нужно задать, чтобы выяснить, полон ли репортаж, над которым он работает).

Он применил 5W в стадии *подготовки требований* проекта создания ИС, предложив такой набор вопросов:

- Кто заказчик этого решения и кому еще оно может быть интересно?
- Каково общее значение решаемой задачи: какое конкурентное преимущество можно получить благодаря этому решению?
- Когда лучше всего установить эту систему и какое значение будут иметь возможные задержки?
- Где именно в организации можно применить это решение?
- Зачем мы предоставляем этот сервис своим пользователям/клиентам?
- Как можно использовать это приложение для других целей?

Но 5W не единственный творческий подход, предлагаемый Кугером. Например, на этапе *логического проектирования* он предлагает применять метод, заимствованный у знаменитого специалиста по решению математических задач Пойа [Polya 1971] и называемый «контрольным листом» (checklist), когда решающий задачу проходит по списку стимулирующих вопросов типа: «Сталкивался ли я с этой задачей раньше? В таком же или в ином виде? Нет ли родственной задачи, которая может оказаться полезной? Есть ли какая-нибудь теория, которую можно применить?»

Для *физического проектирования* Кугер предлагает метод с набором глаголов, с помощью которого можно манипулировать проблемой, рассматривая ее с разных точек зрения. Этот метод «манипулирующих глаголов» [Koberg and Bagnall 1981] предлагает 32 таких глагола, включая следующие:

- Умножать (multiply) – можно ли обобщить какие-то части системы, чтобы обеспечить более широкое ее применение?
- Делить (divide) – можно ли разделить пользователей на группы с аналогичными интересами, чтобы небольшой набор интерфейсов пользователя мог удовлетворить потребности очень разных групп пользователей?
- Исключать (eliminate) – можно ли отказаться от каких-то специфических функций, вначале казавшихся обязательными?

Вот остальные глаголы: подчинять (subdue), обращать (invert), разделять (separate), переставлять (transpose), унифицировать (unify), искажать (distort), поворачивать (rotate), выравнивать (flatten), сжимать (squeeze), дополнять (complement), погружать (submerge), замораживать (freeze), смягчать (soften), взбивать (fluff-up), обходить (bypass), прибавлять (add), вычитать (subtract), освещать (lighten), повторять (repeat), утолщать (thicken), расширять (stretch), выталкивать (extrude), отвергать (repel), защищать (protect), выделять (segregate), интегрировать (integrate), символизировать (symbolize), абстрагировать (abstract), расчленять (dissect).

Программное проектирование порождает еще один творческий подход, который Кугер, основываясь на работе [Crawford 1954], называет «перечислением атрибутов» (attribute listing). При этом разработчик системы перечисляет атрибуты задачи, которую нужно решить, воздерживаясь от их оценки, но манипулируя ими так, чтобы путем модификации найти лучшее решение проблемы.

Интересно, что Кугер завершает эту часть своей работы замечанием о возможной универсальности творчества. Он цитирует Карла Роджерса: «Исходя из своего опыта я пришел к убеждению, что творческое начало есть в каждом человеке, и для его проявления нужны только соответствующие условия». Не вполне понятно, почему Кугер привел эту цитату в конце своей работы, а не в ее начале, рядом с определением творчества, но, возможно, он просто хотел сказать, что те, кто связан с жизненным циклом программы, на любом его этапе вполне могут применять творческие методы, если в этом возникнет необходимость. То есть системные аналитики могут быть творцами, и проектировщики могут быть творцами, и программисты могут быть творцами. Вопрос не в том, МОЖНО ЛИ вызвать и проявить творчество, а в том, КАК это сделать.

Ссылки

COUGER 1990 – «Ensuring Creative Approaches to Information Systems Design», *Managerial and Decision Economics II*, pp. 281-295, 1990; J. Daniel Couger.

COUGER 1996 – «Creativity and Innovation in Information Systems Organizations», Boyd & Fraser Publishing Co., 1996; J. Daniel Couger.

CRAWFORD 1954 – «The Techniques of Creative Thinking», Prentice-Hall, 1954; R.P. Crawford.

KOBERG AND BAGNALL 1981 – «The Universal Traveler: A Soft-Systems Guide to Creativity, Problem-Solving, and the Process of Design» (Revised edition), Crisp Publications, 1991; Don Koberg and Jim Bagnall.

NEWELL AND SHAW 1992 – «The Process of Creative Thinking», in *Contemporary Approaches to Creative Thinking*, edited by H.E. Gruber et al, Atherton Press, 1992; A. Newell, J.C. Shaw, and H.A. Simon.

POLYA 1971 – «How to Solve It», Princeton University Press, 1971; G. Polya.[1]

[1] Пойа Д. «Как решать задачу: пособие для учителей». – Пер. с англ. – М.: Учпедгиз, 1980.

11.2. Творчество и разработка ПО: недостающее звено

В программировании есть две противоположные точки зрения, трение и конфликт между которыми неуклонно усиливаются. Согласно первой создавать программы лучше всего с помощью формального упорядоченного процесса. Согласно второй программы нужно создавать с помощью свободного творческого процесса.

Этот конфликт мы замечаем в докладах на конференциях и статьях по программированию, равно как в методах, применяемых в организациях, где производится программное обеспечение. Когда писалась эта книга, в компьютерной литературе преобладали сторонники формализма. Применение формальных методов считается *обязательным* при создании крупных программных продуктов и *уместным* для ПО любого масштаба. В статьях, посвященных формальным методам, можно встретить такие высказывания, как «...лидеры программного инжиниринга все чаще призывают к более систематическим подходам, требующим больше математики, науки и инжиниринга» [CSTB90]. Согласно этой формалистической точке зрения творческие подходы оказываются контрпродуктивными: они бывают эффективными, но чаще непредсказуемы.

Но в тех же изданиях встречается и противоположная точка зрения. Формальные методы оспариваются все чаще. Например, читаем в [Denning 1991]: «Формализм не является заблуждением. Он продемонстрировал замечательную технологическую силу. Но при этом он ограничивает круг задач, решение которых оказывается доступным для нас. Необходимо выйти за рамки формализма...»

Ярче всего этот конфликт проявляется в области разработки ПО. Формальный упорядоченный подход обычно предполагает следование следующим правилам:

1. Придерживаться методологии жизненного цикла как последовательности выполняемых в определенном порядке этапов работы, каждый из которых завершается выпуском оговоренных документов. В разработке ПО эти этапы называются предварительным, или логическим, проектированием и детальным, или физическим, проектированием.
2. В ходе разработки крупные задачи решаются путем их декомпозиции в виде дерева более мелких и легче решаемых задач. При решении этих мелких задач применяются принципы сильной связности (high cohesion), предполагающей ориентированность решения на связанный круг проблем, и слабого сцепления (loose coupling), предполагающего максимальную автономность каждого решения.

3. Готовый проект представляется с помощью некоторого формального, часто графического, языка. В разные годы в качестве такого языка предлагались UML, блок-схемы, структурные диаграммы и язык проектирования программ (псевдокод).

Сторонники более творческих методов утверждают, что именно здесь, в разработке, недостатки формального подхода наиболее заметны. Хотя методологические этапы и используемое представление являются важными составляющими процесса разработки, говорят они, ясно, что здесь чего-то не хватает. В методологических этапах и технологии представления нет ничего, что в действительности было бы разработкой. Это лишь подготовка к разработке и оформление ее результатов. Спрашивается, где же *собственно* разработка?!

Задавшись таким вопросом, пару десятилетий назад несколько исследователей попытались выяснить, где же на самом деле происходит процесс разработки. Этот поиск называли поиском «недостающего звена» между методологией и представлением, которое и является собственно разработкой.

Результаты этого исследования, на первый взгляд, кажущиеся сегодня устаревшими, тем не менее, остаются актуальными в силу двух обстоятельств:

1. С тех пор никто больше не изучал разработку ПО с этой точки зрения.
2. Сущность разработки ПО за прошедшее время не изменилась.

В поисках отсутствующего звена

В поисках этого отсутствующего звена исследователи обратились к лучшим примерам из практики. В конце концов, разработка ПО успешно ведется уже больше полувека. То есть если в теории разработки и есть недостающее звено, то в практике разработки его нет. Питер Деннинг (Peter J. Denning) [Denning 1991] пишет: «Пора уделить больше внимания туманным, неопределенным, неформализуемым областям повседневной практики, в которых, видимо, и заключается разработка».

Именно так и поступили эти исследователи.

Для изучения практической разработки были выбраны два подхода. В одном случае опрашивали ведущих разработчиков-практиков. В другом случае записывали на видео реальные сеансы разработки, и по процессу, применяемому этими разработчиками, создавали его теоретическое описание. Несмотря на отличие подходов, их результаты оказались удивительно согласованными, о чем будет сказано ниже.

В подходе с опросом, в свою очередь, были применены два довольно разных метода. Ветеран программирования Джон Нестор (John Nestor), работавший в то время в SEI, определил десять ведущих, по его мнению, разработчиков ПО исходя из своего опыта работы с ними и опросил их, доложив о результатах на внутреннем совещании в SEI. А журналистка Сьюзен Лэммерс (Susan Lammers) опросила ряд успешных предпринимателей в области программ для микрокомпьютеров и опубликовала результаты в книге [Lammers 1986]. Здесь приводится краткая сводка результатов, полученных Нестором; о результатах Лэммерс будет сказано далее.

Что же выяснил Нестор у своих выдающихся разработчиков? Он задал им несколько вопросов, в том числе «что такое проектирование» и «какими качествами должны обладать выдающиеся разработчики». Ответы звучали примерно так:

- У выдающихся разработчиков есть большой комплект стандартных шаблонов.
- У выдающихся разработчиков бывали неудачи, из которых они сделали выводы.
- Выдающиеся разработчики в совершенстве владеют профессиональными инструментами.
- Выдающиеся разработчики больше склоняются к простоте.
- Выдающиеся разработчики предвидят многие изменения.
- Выдающиеся разработчики умеют ставить себя на место пользователя.
- Выдающихся разработчиков не пугает сложность.

Конечно, столь субъективное исследование всегда небезупречно. Выбор разработчиков и выводы целиком определялись точкой зрения на проектирование одного единственного человека. Тем не менее, в результатах Нестора есть очень интересные факты. Очевидно, что выдающиеся разработчики, обладая некоторым опытом, в том числе неудачным, сумели извлечь из этого опыта ряд моделей, которые применяли в последующих проектах. (Так и вижу за их плечами рюкзаки, полные аналогий, куда можно залезть, чтобы подыскать отправную точку для новой разработки. Чем полнее рюкзак, тем успешнее разработка.) Успешные разработчики могут также браться за крайне сложные задачи и находить для них простые решения. Несмотря на субъективность подхода, полученные Нестором результаты несомненно важны.

Метод видеозаписи

Второй метод исследования сущности разработки путем изучения практики, когда ведется видеозапись, позволяет избежать субъективности. Процедура была организована так:

1. Отдельных разработчиков или их группы записывают на видео в процессе разработки решения для поставленной задачи.
2. Во время отдельных сеансов работы разработчикам предлагается «думать вслух», чтобы можно было зарегистрировать процесс размышления.
3. Видеозапись неизбежно оказывает влияние на работу разработчиков, но организована так, чтобы мешать им как можно меньше.
4. Результаты нескольких видеосеансов с разными группами разработчиков изучаются с целью обнаружить одинаковые черты и виды деятельности.
5. Результаты указанного выше изучения теоретически обобщаются.
6. Правильность теоретических обобщений проверяется путем сравнения с новыми записанными сеансами.
7. Теоретическое обобщение предъявляется участникам сеанса разработки, которые выражают свое мнение о том, правильно ли описан процесс, который они применяли.

Такая последовательность действий называется «протокольным анализом» и применяется во многих видах исследований, связанных с наблюдением. Исследователями, применившими его при изучении процесса разработки ПО, были Эллиот Солоуэй (Elliot Soloway), в то время сотрудник Йейльского университета, а впоследствии Мичиганского, и Билл Кертис (Bill Curtis), в то время сотрудник Microelectronics and Computing Consortium (MCC), а впоследствии SEI и Borland. Результаты исследования были опубликованы в конце 1980-х [Curtis 1987, Guinon 1987, Adelson 1984, Adelson 1985, Soloway 1987].

Что фактически обнаружили эти исследования сущности разработки? Результаты можно разделить на два уровня.

На первом уровне исследователи выяснили, что в разработку входят:

1. Понимание задачи.
2. Декомпозиция задачи на цели и объекты.
3. Подбор и составление планов решения задачи.
4. Выполнение этих планов.
5. Размышление над результатом и процессом разработки.

Результаты, полученные на этом уровне, были интересны, но особых открытий не содержали. В действительности, они даже подвергались критике на том основании, что описывали общий процесс решения задач, а не конкретно процесс разработки. Эта критика представляется обоснованной: если взглянуть объективно, то, эти шаги представляют собой иную формулировку идеи жизненного цикла ПО, хорошо проработанную задолго до данного исследования. Если бы исследование осталось на этом уровне, его результаты остались бы малозаметными.

Золотая жила

Но был еще один уровень, на котором исследователи наткнулись на золотую жилу. Они подвергли более доскональному исследованию проблему «отбора и составления планов», чтобы узнать, какие виды деятельности предполагает эта задача. Тут они и обнаружили то, что теперь принято считать сутью творческого проектирования.

Как выяснилось, при составлении планов разработки разработчики делают следующие шаги:

1. Строят мысленную модель предположительного решения задачи.
2. Мысленно проверяют работу этой модели, чтобы узнать, действительно ли она решает задачу. Часто такое прорабатывание в уме (называемое также «имитацией») происходит в виде подачи каких-либо данных на вход и проверки корректности получаемых при этом результатов.
3. Если выходные данные оказываются некорректными (а это часто случается на ранних стадиях разработки), то модель развивают с целью исправить недостатки, а затем проверяют снова.
4. Когда выходные данные оказываются, наконец, корректными, шаги 2 и 3 повторяются с другим набором входных данных.
5. Когда достаточное количество наборов входных данных прошло проверку на шаге 4, модель признается пригодной моделью разработки, и начинается этап представления проекта.

В этой процедуре важно отметить две вещи. Во-первых, весь процесс чисто когнитивный, то есть все действия проводятся в уме, со скоростью мышления. Это существенно, поскольку успех разработки зависит от этой скорости. Карандаш и бумага или даже компьютер на этой стадии существенно замедляют, а потому ухудшают процесс разработки.

Во-вторых, это итеративный процесс. Это означает, что разработчик повторяет определенные этапы, используя метод проб и ошибок, пока не будет найдена предположительно правильная конструкция. Исследователи называют такой метод проб и ошибок «эвристическим» методом.

Особенно важно, что оба эти аспекта отчасти противоречат более формальным методологическим подходам. Поскольку все эти этапы процесса разработки происходят в уме, трудно (и вредно) подчинить их какому-либо формальному упорядоченному процессу. Будучи эвристичными, они представляют собой почти полную противоположность формальным методам (формалисты часто выдают эвристические методы за «ошибочную» противоположность их любимым формальным методам). Таким образом, здесь, в когнитивной сущности разработки, мы обнаружили то место, где формальные методы попросту оказываются неэффективными.

Это открытие не стало сюрпризом для тех, кто уже давно выражал беспокойство по поводу формальных методов. Например, Х. Дитеру Ромбаху (H. Dieter Rombach) принадлежат слова [Rombach 1990]: «Хотя формализация... оказывается решением для достаточно механических процессов,... для процессов разработки она непригодна». Теперь мы видим, почему.

Вернемся на время к другому подходу к изучению проектирования – с проведением опросов. Как мы помним, результаты ряда опросов нашли отражение в книге [Lammers 1986]. Вот некоторые интересные цитаты из этой книги.

> «Программирование начинается с воображения. Я стараюсь кристально ясно представить, как это должно выглядеть. На этой начальной стадии я беру карандаш и бумагу. Просто машинально что-то вычерчиваю,... потому что настоящая картина находится в моем мозгу» (Чарльз Симони, разработавший табличный процессор Multiplan).
>
> «В какой-то момент [проект] взрывается, и я сразу все вижу внутренним зрением... В мозгу происходят чудесные вещи, но я ничего не могу записать, потому что постоянно все меняю» (Гэри Килдалл, разработчик операционной системы CP/M).
>
> «Нужно смоделировать в голове работу программы... Когда вы что-то создаете... и представляете модель, необходимо одиночество» (Билл Гейтс, основатель и CEO Microsoft).
>
> «Вы все время стараетесь сохранить состояние всей системы, над которой работаете, у себя в мозгу. Если потерять этот мысленный образ, вернуться в то же состояние можно нескоро» (Джон Пейдж, разработчик PFS:File).

Просто поразительно, насколько эти результаты опросов созвучны результатам протокольного анализа видеозаписей процесса! Участники опросов живыми словами выразили более сухие аналитические результаты. Особенно интересно, что участники опроса использовали такие слова, как «модель», «моделирование», и говорили об «одиночестве» творца-разработчика, который во время разработки должен не потерять

«мысленный образ». Очевидно, что это описание творческого, а не формального и упорядоченного процесса.

Эмпирическое исследование

Исследования, основанные на изучении практики, обычно называются «эмпирическими». Результаты многих таких исследований первоначально были представлены на ряде конференций, посвященных эмпирическим исследованиям и называвшимся «Empirical Studies of Programmers». (Сторонники формальных методов не преминули обратить внимание на неудачный выбор акронима – ESP[1]!) В последнее время эмпирические результаты публикуются в специальном журнале «Empirical Software Engineering» издательства Шпрингера. Необходимо отметить, что результаты эмпирического исследования далеко не ограничиваются открытием эвристических методов.

Например, исследователи ESP отошли от деталей когнитивного процесса разработки, чтобы рассмотреть более общие его элементы. Они выяснили, что разработка состоит из нескольких компонентов, ориентированных:

- на решаемую задачу;
- на природу и понятие собственно разработки;
- на роль человека-разработчика;
- на программную поддержку труда разработчика.

При рассмотрении этого списка компонентов разработки ряд исследователей ESP постепенно убеждается в том, что сложность задач разработки связана не с самим процессом разработки, а с разработкой решения в контексте конкретной прикладной задачи. Например, разработчик должен обладать не только определенными базовыми сведениями о прикладной области (проблемно-ориентированными), но и специальными сведениями о методах разработки, применяемых именно в этой области (проектно-ориентированными). Например, разработчики генераторов отчетов должны знать, как генерировать цифровые поля в строках итогов с несколькими уровнями подытогов, а разработчики программ для научных исследований должны знать, как итеративные математические методы реализуются на компьютерах, где точность представления чисел ограничена.

---

[1] По-другому ESP можно расшифровать как extrasensory perception, то есть экстрасенсорное (сверхчувственное) восприятие. – *Прим. перев.*

Разработчик как человек

Вероятно, самыми захватывающими компонентами процесса проектирования являются возможности и ограничения, связанные с личностью разработчика.

Вспомним, что сердцевину процесса разработки составляет когнитивная эвристическая деятельность. Какие преимущества и недостатки человеческого разума влияют на эту деятельность?

Прежде всего, психологи установили, что человеческую память можно разделить на две части – «кратковременную» и «долговременную». (Здесь можно провести аналогию с оперативной и внешней памятью компьютера.) Кратковременная память действует очень быстро, но ее объем ограничен; долговременная память действует чуть более медленно, но ее объем представляется практически неограниченным. На самом деле, кратковременная память существенно ограничена по объему: исследования, начавшиеся еще в 1950-х, обнаружили, что в этой области памяти можно хранить одновременно лишь от пяти до девяти разных элементов. (У психологов это называется «магическое число семь плюс-минус два».)

Поскольку когнитивная разработка происходит со скоростью света, она должна выполняться в кратковременной памяти, но волшебное число 7 жестко ограничивает разработчика. К счастью, если верить психологам, мозг придумал, как обойти это ограничение. Он применяет «разбивку на блоки», объединяя несколько предметов в один «блок», и мозг может удержать до семи *таких* блоков. (Например, узнавание человека по лицу происходит не путем запоминания набора признаков низкого уровня этого лица, таких как «форма губ», «цвет глаз», «волосяной покров», а путем формирования некоторого образа или сводного объекта, который можно назвать «лицо».) Так же и разработчики могут обойти ограничение человеческой кратковременной памяти путем разбиения на блоки. И все же ограниченность памяти остается серьезной проблемой для разработчика. Вспомним, например, об одиноком разработчике, который не может прерваться из-за опасения потерять образ проекта. Здесь проявляются ненадежность и ограниченность кратковременной памяти: потерянный из кратковременной памяти образ нельзя восстановить, и проблема усугубляется тем, что кратковременная память содержала не группу простых фактов, а несколько сложных информационных блоков.

Эмпирические исследования привели к еще двум важным открытиям, касающимся разработки. Первое относится к началу разработки. А второе – к ее концу.

Начало разработки

Одним из главных открытий исследователей ESP стало обнаружение того, что разработчики при любой возможности начинают разработку с повторного использования уже имеющегося проекта. (Вспомним «рюкзак аналогий» за плечами разработчика, в котором он ищет старый проект, оставшийся от аналогичной задачи).

Первой моделью в эвристическом процессе когнитивной разработки вполне может оказаться какая-то уже применявшаяся ранее, а не новая, созданная для данной конкретной задачи. Обнаружив, что если разработчик уже встречался с подобной задачей, то за отправную точку он выбирает «типовую программу», Виссер заметил, что «разработчики редко начинают с чистого листа» [Visser 1987]. То же самое мы встречаем в одном из опросов в [Lammers 1986], где заслуженный разработчик программ для мэйнфреймов и микрокомпьютеров Батлер Лэмпсон (Butler Lampson) заявил: «В большинстве случаев новая программа оказывается уточнением, расширением, обобщением или улучшением уже существующей. Весьма редко то, что делаешь, оказывается совершенно новым...»

Есть еще один интересный эмпирический результат, касающийся разработки. Хотя в учебниках и литературе разработку часто называют нисходящим процессом, он редко оказывается таким на практике. Билл Кертис сказал, что «в отсутствие внешнего воздействия процесс разработки оказывается гибким» – имеется в виду, что хорошие разработчики действуют не упорядоченно, а по хаотической схеме, возникающей у них в мозгу вслед за новыми возможностями, а не в заданной последовательности. Некоторые считают, что в действительности разработка происходит в порядке убывания сложности, то есть сначала разработчик определяет подзадачи, которые кажутся ему наиболее сложными, и занимается в первую очередь ими, чтобы застраховаться от возможных проблем, если трудность решения какой-либо подзадачи поставит под угрозу все его решение. Любопытно, что известный компьютерный специалист Дэвид Парнас рекомендовал «подделывать» нисходящий процесс, то есть даже если разработка происходит не по нисходящей схеме, регистрация и представление разработки *должны* иметь нисходящий вид, поскольку только в этом случае другие смогут разобраться в проекте.

Здесь повторяется та же история: «оппортунистическая» разработка и правило «сначала сложное, потом простое» находят отражение в опросах, приводимых в [Lammers 1986]:

> «Начинаешь с той проблемы, которая кажется самой трудной для решения, а потом разбиваешь ее на более мелкие части», – говорит Гэри Килдалл (Gary Kildall), разработчик CP/M.

> «Справившись со сложными задачами по отдельности – возможно, написав какую-нибудь программку, чтобы проверить свою теорию, – я уже могу структурировать программу», – говорит Джон Пейдж (John Page), создатель PFS:File.

Завершение разработки

Если, как мы видели, в начале разработки обнаружились такие отклонения от общепринятых представлений, как повторное использование прежних проектов и оппортунистический подход с приоритетом трудных подзадач, то принесло ли какие-нибудь результаты исследователям ESP изучение завершения разработки? Да, здесь их тоже ждали некоторые сюрпризы.

В учебниках пишут, что нисходящая разработка ведется до того уровня, на котором знающий программист уже сможет писать код. Но что это означает в действительности? У лучших разработчиков есть четко определенные критерии для окончания разработки: процесс завершается, когда то, что им осталось разработать, аналогично задачам, которые они решали раньше, и они делали это столько раз, что кодирование не требует привлечения творческого мышления. Такие готовые решения называются «примитивами». Естественен вопрос: одинаковы ли наборы примитивов у разных разработчиков?

Ясно, что ответ отрицательный. Примитивы возникают у разработчика благодаря не только навыкам разработки, но и (как мы уже видели в этой статье) знанию одной или нескольких прикладных областей и специфичных для них методов разработки. Таким образом, разработчик *A*, работая над генератором отчетов, может остановиться на довольно высоком уровне, если он ранее разрабатывал и кодировал много таких же задач, тогда как разработчик *B*, у которого опыта не меньше, чем у *A*, но писать генераторы отчетов ему не приходилось, может довести разработку до более детального уровня. Конечно, если код будет писать тот же, кто разрабатывал проект, проблем не возникнет. Однако все чаще – особенно в крупных системах – происходит передача проекта от его разработчика к кодировщику. Если наборы примитивов разработчика и кодировщика совпадают, такая передача происходит гладко.

Но вот два случая, когда наборы примитивов не совпадают:

- У разработчика более мощные примитивы, чем у кодировщика: тогда разработчик завершает работу на относительно высоком уровне, в результате чего кодировщику придется выполнить дополнительную проектную работу, прежде чем он сможет начать писать код.

- У кодировщика более мощные примитивы, чем у разработчика: тогда кодировщик может возмутиться излишней детализацией предлагаемого проекта (которая может быть ошибочной или более сложной, чем сделал бы он сам), и тогда он просто выкинет лишние части проекта.

Суть в том, что завершение разработки происходит совсем не так просто, как описываемая в учебниках передача проекта для реализации «знающему программисту». На практике часто оказывается, что после передачи происходит дальнейшая разработка или отбрасывание части проекта.

Есть еще одна опасность, связанная с завершением разработки. Природа этого процесса такова, что временами на самом нижнем уровне разработки таится какая-нибудь столь сложная проблема, что вся созданная конструкция оказывается неосуществимой, и разработчик вынужден отказаться от нее и начать все сначала. Это тот случай, когда разработчик проглядел «сложную часть» проекта, оставив ее на конец работы, а то, что он наработал к этому моменту, не позволяет справиться со сложной частью задачи. Большинство разработчиков признают, что им случалось сталкиваться с такой проблемой, но нет никаких исследований, в которых делалась бы попытка выяснить тип проблем, вызывающих такой эффект, или найти признаки того, что такие проблемы возможны.

И снова эти мысли подтверждаются цитатами из [Lammers 1986]: «Довести разработку до уровня,... на котором выясняются две вещи. Во-первых, я уже знаю примитивы, которыми описываю программирование – они неоднократно применялись ранее... Во-вторых, я достаточно хорошо знаю эти примитивы, чтобы примерно оценить, сколько памяти они потребуют... Я могу быть достаточно уверенным в том, что эту функцию можно реализовать, и могу примерно оценить ее эффективность. Конечно, и в этом случае можно проглядеть что-нибудь весьма существенное, но тут уж ничего не поделаешь», – говорит Батлер Лэмпсон (Butler Lampson).

Выводы

Можно ли определить сущность разработки ПО как формальный или творческий процесс? Конечно, мы должны признать, что формальные процессы методологий и представлений – большое подспорье для разработки, но столь же несомненно, что они не являются ее *сутью*.

Суть разработки, как выяснили исследователи, изучавшие лучшие практические примеры разработки, характеризуют следующие черты:

1. Эвристический когнитивный процесс, не допускающий формальных подходов.
2. Наличие опыта создания повторно используемых конструкций, которые можно брать за основу нового проекта.

3. Оппортунистический, а не нисходящий подход к этапам разработки.
4. Имеющий индивидуальные особенности и определяемый набором примитивов процесс определения момента завершения разработки.
5. Возможность выявления скрытых сложных задач, способных в последнюю минуту разрушить практически завершенный проект.

Такие выводы могут огорчить тех, кто считает, что программное обеспечение следует создавать только на основе формальных методов, но понимание необходимости дополнить формальные процессы творческими позволит углубить представление о том, как выпускать программные продукты более высокого качества.

Ссылки

ADELSON 1984 – «A Model of Software Design», report from the Dept. of Computer Science, Yale University, October 1984; B. Adelson and E. Soloway.

ADELSON 1985 – «The Role of Domain Experience in Software Design», *IEEE Transactions on Software Engineering,* October 1985; B. Adelson and E. Soloway.

CSTB 1990 – «Scaling Up: A Research Agenda for Software Engineering», a report of the Computer Science Technology Board of the National Research Council,» *Communications of the ACM,* March 1990.

CURTIS 1987 – MCC Technical Report Number STP-260–87, «Empirical Studies of the Design Process: Papers for the Second Workshop on Empirical Studies of Programmers», September 1987; B. Curtis, R. Guindon, H. Krasner, D. Walz, J. Elam, and N. Iscoe.

DENNING 1991 – «Technology or Management? An Editorial», *Communications of the ACM,* March 1991; Peter J. Denning.

GUINON 1987 – MCC Technical Report Number STP-283–87, «A Model of Cognitive Processes in Software Design: An Analysis of Breakdowns in Early Design Activities by Individuals», August 1987; R. Guindon, B. Curtis, H. Krasner.

LAMMERS 1986 – «Programmers at Work», Microsoft Press, 1986; Susan Lammers.

ROMBACH 1990 – «Design Measurement: Some Lessons Learned», *IEEE Software,* March 1990; H. Dieter Rombach.

SOLOWAY 1987 – «E Unum Pluribus: Generating Alternative Designs», Dept. of Computer Science, Yale University, 1987; E. Soloway, J. Spohrer, D. Littman.

VISSER 1987 – «Strategies in Programming Programmable Controllers: A Field Study on a Professional Programmer», *Proceedings of the Empirical Studies of Programmers, Second Workshop,* Ablex, 1987; Willemien Visser.

11.3. Творчество и реальность: конкретный случай

Вы уже знакомы с проф. Дж. Дэниелом Кугером по предыдущим главам этой книги. Поэтому для вас не должно оказаться неожиданностью, что проф. Кугер решил применить в какой-нибудь реальной организации то, что ему удалось узнать о творчестве и процессе создания программного обеспечения, и исследовать этот конкретный случай.

Ниже приведены посвященные исследованию конкретного случая отрывки из его книги «Creativity and Innovation in Information Systems Organizations» (Творчество и новаторство в организациях, использующих информационные системы) [Couger 1996]. Организацией, где он провел такое исследование, стал Объединенный технологический центр микроэлектроники (United Technologies Microelectronics Center, UTMC), расположенный в том же городе, что и университет, в котором работал проф. Кугер, – Университет Колорадо в Колорадо-Спрингс (University of Colorado at Colorado Springs, UCCS). UTMC специализируется в основном на выпуске высоконадежной военной и аэрокосмической техники, а то подразделение, в котором проводилось исследование, занималось разработкой и сопровождением компьютерных программ для корпоративного управления UTMC (например, научно-техническими приложениями вроде автоматизированного программирования оно не занималось – за них отвечало другое подразделение).

Когда начиналось это исследование [Couger, McIntyre, Higgins and Snow 1991], руководство UTMC рассматривало соответствующее подразделение как «весьма эффективное». Считалось, что оно выше среднего уровня, а иногда просто лидирует по таким показателям, как выполнение обязательств, подготовка пользователей, соблюдение графиков, количество допускаемых ошибок, решение задач, легкость эксплуатации и предоставление информации о состоянии дел. Проф. Кугер рассудил, что если удастся продемонстрировать пользу программы стимуляции творчества для организации, уже добившейся успеха с ИС, то другим организациям такая программа может показаться еще привлекательнее.

Было решено действовать в двух направлениях:

1. Провести организационные мероприятия для стимуляции творчества и новаторства.
2. Провести обучение специальным технологиям порождения и оценки творчества.

Сначала взялись за обучение. Ниже вы увидите, какие предметы были выбраны. Они составят прекрасную основу для любой организации, которая захочет применить такие же методы.

На втором этапе предполагалось подкрепить обучение практическим применением полученных идей и обсудить результаты этого применения.

Посмотрим сначала, какие темы были взяты для изучения. Приведенный ниже список развивает темы из предыдущего раздела, перечисляя приемы, составляющие каждый этап жизненного цикла ПО. На основе сделанного проф. Кугером анализа литературы о творчестве и его представлений о применимости всего этого в области информационных систем для обучения программистов в UTMC творческим методам была составлена следующая учебная программа.

Описание творческих методов

Аналогия/метафора. Аналогия – это утверждение о том, что некие объекты, люди, ситуации или действия имеют сходство развития или взаимоотношений. Метафора – это фигура речи. С помощью метафор и аналогий можно создавать воображаемые ситуации, позволяющие по-новому взглянуть на сущность задачи и ее решение.

Мозговая атака. Цель этого метода – порождение множества идей за короткий промежуток времени с помощью процесса, стимулирующего генерацию идей. Генерация идей отделяется от их оценки, чтобы поощрить воображение.

Голубые карточки. Каждый записывает свою идею на голубом листке размером 3×5 дюймов. Имена авторов обсуждаемых идей не указываются, что позволяет высказывать их более свободно. Когда каждая идея записана на отдельной карточке, их проще сортировать и группировать.

Экстраполяция. Применение проверенного метода или приема к новой задаче.

Метод последовательных абстракций. Выработка альтернативных описаний задачи путем последовательного перехода на более высокие уровни абстракции, пока не будет получено удовлетворительное определение. Когда с помощью такого способа осуществляется систематическое укрупнение задачи, возникают ее новые определения, пользу и возможность решения которых можно оценить. По достижении подходящего уровня абстракции легче увидеть возможные решения. Большое достоинство этого приема – предоставление аналитику опре-

деленной структуры задачи, позволяющей систематически изучать вложенные структуры и связи.

Метод 5W и H. Вопросы кто-какой-где-когда-зачем-как помогают расширить видение задачи и проверить, все ли ее необходимые стороны рассмотрены.

Полевой анализ действующих факторов (Force Field Analysis Technique). Название отражает возможность выявлять с помощью этого метода силы, способствующие или препятствующие решению задачи. Творческое мышление стимулируют три пути: 1) определение направления (предвидение), 2) выявление поддерживающих факторов, которые можно усилить, и 3) выявление ослабляющих факторов, которые можно подавить.

Мирные условия. Цель этого метода – помочь людям мысленно отстраниться от окружающей обстановки и способствовать более спокойному и открытому мыслительному процессу. Они представляют себя в приятных условиях, где можно расслабиться, – на берегу моря или на горном склоне.

Обращение задачи. Иногда бывает полезно обратить постановку задачи, чтобы проанализировать ее в другой среде. Например, чтобы улучшить творческий климат, можно рассмотреть противоположную задачу: как «убить» творчество.

Метод ассоциаций/образов. В основе лежит природная склонность человека находить связи между вещами. Процесс связывания и объединения служит еще одним способом расширить пространство решений.

Принятие желаемого за действительное. Этот метод особенно полезен для тех, кто склонен излишне аналитически подходить к решению задач. Он позволяет ослабить допущения и рассмотреть более широкий, чем обычно, набор альтернатив. Метод требует пофантазировать в процессе решения, что может привести к неожиданным подходам. Выбор необычной исходной точки для решения задачи иногда открывает такие перспективы, которые иначе не могли возникнуть.

После обучения и применения этого комплекса методов в UTMC проанализировали достигнутые результаты. Конечно, это был достаточно субъективный анализ (когда сталкиваются точность исследования и его актуальность, в полной мере удовлетворить то и другое бывает трудно. Исследования конкретных случаев часто делают уступку точности в пользу актуальности, не было исключением и данное исследование.)

Профессор Кугер провел опрос, чтобы выяснить мнения участников. По их предположениям, выигрыш составил:

- для каждого из участников (по личной оценке) – около 23%;
- для подразделения – около 27%.

Но личными оценками участников результаты проф. Кугера не ограничились. Каждый из работников вел «творческий дневник», в который записывал сделанное им благодаря приобретению новых знаний. В этих дневниках обнаружилось 12 конкретных улучшений в эффективности разработки и еще 6 улучшений в продуктивности:

- сокращение времени на расширение базы данных с 60 до 4 часов;
- сокращение времени обсчета текущего года с 24 часов до 45 минут;
- сокращение обработки прогноза заказов с 6 часов до 5 минут;
- сокращение времени на сопровождение программ на 10–15 часов в месяц.

Такие весьма заметные достижения заставили президента UTMC попросить обучить этим методам и других работников.

Приведенные выше цифры весьма впечатляют, но в то же время остаются весьма нейтральными эмоционально, поэтому мы опишем конкретные улучшения, на которые указали сотрудники ИС UTMC:

Творческие усовершенствования в отделе ИС UTMC

Следующие описания составлены аналитиками и программистами; сохранено первое лицо «мы», чтобы показать, как гордятся сотрудники творческими достижениям и своей команды.

1. **Модель предприятия.** Подразделение АСУ изучало подходы к созданию модели предприятия для компании. Пока мы обсуждали различные подходы с поставщиками и консультантами, глава UTMC посетил семинар, где узнал о семи методах планирования для решения бизнес-задач. Он хотел внедрить эту технологию в UTMC. Отдел АСУ адаптировал технологию планирования с помощью вопросов вида «что, если...» и включил ее в общую методологию модели предприятия. Мы также отошли от обычной процедуры моделирования текущего бизнеса, предпочтя моделировать «идеальный» бизнес. Этот метод должен показать имеющиеся возможности совершенствования бизнес-процессов UTMC. Отдел АСУ обучает этим методам менеджеров UTMC, которые примут участие в моделировании предприятия.
2. **Реорганизация автоматизированного производства.** Ранее в этом году мы вместе с отделом управления производством участвовали в проекте модернизации модуля планирования, входящего в нашу

покупную систему управления производством (computer-aided manufacturing, CAM). Эта система появилась в UTMC семь лет назад, когда мы занимались совершенно другим. Например, у нас не было собственного производства кристаллических плат, и наша продукция предназначалась только для UTC. Сейчас мы производим продукцию не только для потребителей в UTC, но и на продажу, а это весьма разные вещи. Поэтому нам трудно было заставить модуль планирования правильно работать с теперешней структурой поставок и производства. Мы решили не пытаться заставить систему делать то, что нам требуется, а взглянуть шире и полностью изменить организацию системы CAM, чтобы она соответствовала нашим задачам в 1990-е годы. Мы применили метод «5W и H» и мозговую атаку и организовали опрос всех подразделений, пользующихся системой CAM. Мы разработали трехлетний план существенного обновления нашей системы для решения задач, поставленных перед компанией на 1990-е годы. Мы представили план высшему руководству, получили его одобрение и приступили к работе над проектом.

3. **Оценка CASE-средств.** Отделу АСУ потребовалось оценить несколько CASE-инструментов (средств автоматизации программирования), которые могли бы повысить продуктивность и качество программного обеспечения. Мы применили творческий метод «обращения задачи» и составили перечень факторов, отрицательно влияющих на продуктивность и качество работы АСУ. В результате мы легко нашли главные факторы, влияние которых можно было бы уменьшить в результате применения CASE-средств, и составили сравнительную матрицу качеств CASE-инструментов, с помощью которой сузили оцениваемый набор до трех инструментов.
4. **Расширение базы данных CAM.** В нашей базе данных CAM есть все данные, относящиеся к производству, и она постоянно расширяется. Для того чтобы поддерживать высокую скорость доступа к базе данных, ее нужно периодически реорганизовывать. Раньше мы выгружали базу данных, вносили изменения и загружали данные обратно в базу. Эта процедура занимала около 60 часов. Руководитель одного из подразделений АСУ искал способы улучшить эффективность системы. Он обнаружил новый способ изменения размера базы данных, для которого не требуется выгрузка/загрузка базы. Мы сумели применить этот способ в наших условиях, в результате чего расширение базы данных стало происходить за четыре часа (экономия в 46 часов). Это хороший пример экстраполяции, когда готовый метод решения применяется в некой новой области.

5. **Автоматическая чистка отчета для главной бухгалтерской книги.** Структуру нашего отчета для главной бухгалтерской книги приходилось приводить в порядок, потому что в ней содержались устаревшие элементы, часть элементов отсутствовала, а часть итогов подсчитывалась неправильно. Финансовое отделение собиралось вносить исправления вручную. Эта процедура заняла бы у них несколько месяцев. Взамен мы предложили воспользоваться методом «принятия желаемого за действительное», чтобы финансисты смогли определить, какой они хотят видеть структуру отчета. Затем мы написали компьютерную программу, которая автоматически приводила структуру отчета в порядок. В результате было удалено более 40 000 устаревших структурных элементов, добавлено более 10 000 отсутствующих и исправлено 2505 ошибок вычисления итоговых сумм. Автоматизация этого процесса не только позволила сберечь много месяцев труда финансистов, но также гарантировала согласованность отчетов для UTC, UTMC и государственных органов.
6. **Использование другой РБД взамен INGRES.** В настоящее время в большинстве наших финансовых систем используется реляционная БД (РБД) INGRES; мы также разрабатываем на INGRES большую маркетинговую систему. Нам нравится гибкость, которую дает эта РБД, но ее лицензирование для нескольких VAX-систем обходится дорого. Сейчас у нас есть кластер VAX, но INGRES установлена только на одной VAX 8650. С увеличением количества приложений, работающих под INGRES, придется устанавливать ее на новые машины. Дополнительные лицензии на INGRES обошлись бы нам недешево. Мы задавали вопросы типа «что, если...» (метод «5W и H») и применили мозговую атаку, в результате чего возникло предложение, которое, возможно, позволит экономичным образом использовать имеющиеся установки INGRES на нескольких машинах. У DEC есть РБД, движок которой бесплатно поставляется со всеми машинами VAX. Клиентские средства INGRES могут работать с движком РБД DEC. Мы собираемся проверить, позволит ли комбинация INGRES+РБД всем нашим приложениям для INGRES выполняться на всех VAX-машинах в кластере.
7. **Рассылка отчетов автоматизированной системы управления.** Отчеты САМ автоматически генерируются раз в месяц, раз в неделю или ежедневно в зависимости от предъявляемых требований. Система определяет частоту генерации отчетов с помощью фиксированных управляющих процедур, в которых определены время составления отчета и его получатель. Такая процедура требует, чтобы один из программистов корректировал ее в соответствии с текущими требованиями. С помощью вопросов «что, если...» (метод «5W

и Н») была разработана система на базе файлов. Мы создали файл, в котором есть вся информация, необходимая для запуска генератора отчетов. В результате удалось сократить время сопровождения на 10–15 часов в месяц. Кроме того, пользователи видят теперь, кто и какие отчеты получает, что особенно полезно при уходе сотрудников в отпуск.

8. **Система заказа STAR/управление отчетами.** Мы разрабатываем новую систему заказа товаров, которая называется STAR. При этом нам потребовалось ввести в нее некоторые дружественные пользователю функции, такие как возможность непосредственного перехода от одной функции к другой, минуя структуру меню. Нам также потребовался удобный клиент для генератора отчетов, облегчающий задание критериев и выбор условий сортировки отчетов. Мы провели мозговую атаку с участием главных аналитиков MIS из разных групп, чтобы найти решение, которое легко сопровождать. Чтобы прийти к окончательному решению, мы много раз задавали вопрос «как нам добиться...». Мы разработали несколько вспомогательных инструментов, применимыми в любом приложении INGRES.
9. **Отчеты об управлении программами.** В отдел АСУ обратился пользователь с просьбой помочь ему с программой для ПК, в которой он вручную вел учет данных об основных программах UTMC. Применив творческий метод «принятия желаемого за действительное», мы выяснили, что ему нужен отчет, который будет извлекать данные из главной бухгалтерской книги, учета трудозатрат и системы управления программами Viewpoint. Мы смогли удовлетворить его «желание», воспользовавшись генератором отчетов UDMS (User Data Management System). Впервые наша компания смогла принимать важные решения на основе обобщенных данных о программах UTMC.
10. **12-месячное окно.** UTMC потребовалось ввести в бизнес практику, разрешающую регистрировать заказ, только если желательная для покупателя дата поставки находится в пределах 12 месяцев. Предыдущая попытка реализовать такую возможность оказалась неудачной, поскольку в имеющейся системе ввода заказов было трудно реализовать соответствующую логику программирования. С помощью метода мозговой атаки с участием старших аналитиков АСУ мы попробовали выяснить, чем была вызвана прежняя неудача и как избежать ее на этот раз. В результате появилась надежная конструкция с совершенно иной файловой структурой.
11. **Система прогнозирования.** UTMC потребовалось, чтобы сотрудники отдела продаж могли обновлять свои прогнозы на месте для увеличения точности и временной привязки прогнозов. Ресурсы были ра-

нее выделены другим проектам, что препятствовало существенной модификации системы прогнозирования с помощью обычных процедур. Тогда мы прибегли к созданию прототипов в РБД INGRES и генераторе отчетов UDMS, затем заключили двухнедельный контракт с программистом, который разработал промежуточное решение, действующее во время выполнения существенной модификации. Таким образом, UTMC получила временное решение; также были получены ценные данные для выработки требований к конечной системе. Метод «что, если...» в совокупности с установкой на творческую среду позволил объединить усилия АСУ, отделов маркетинга, продаж и поставщика программного обеспечения, с которым мы заключили контракт.

12. **Интерфейс пользователя для прокрутки в INGRES.** В INGRES есть встроенная возможность прокрутки, которая действует непривычным для пользователя образом. Вместо того чтобы перемещаться по полям с помощью клавиши TAB, пользователь должен нажимать Enter. Это недружественная пользователю особенность, и она затрудняет пользование системой. С помощью вопросов «что, если...» мы внесли полезное изменение в наш код INGRES, позволившее работать с прокруткой привычным способом. Это изменение сделало систему STAR более дружественной, уменьшив вероятность ошибок при вводе.
13. **Расходы за истекший год (YTD).** UTMC потребовалось подсчитывать расходы за истекший год (Year-to-Date, YTD) во время обычного цикла закрытия, но поскольку программе требовались для этого целые сутки, пришлось бы отложить выполнение всех других важных задач. Отделу АСУ пришлось пересмотреть метод подсчета YTD и предложить новое решение. Применив метод «5W и H», мы выяснили, что можем перевести две крупные программы, написанные на COBOL, на языки DCL (Digital Command Language) и SQL (Structured Query Language), в результате чего время работы сократилось до 45 минут.
14. **Отчет по плану заказов.** Поскольку получение отчета по плану заказов обычно занимает 6 часов, эти данные было невозможно своевременно использовать в бизнес-цикле прогнозирования. Чтобы найти другое решение, мы воспользовались творческим методом «5W и H». В результате было предложено использовать INGRES вместо файлов RMS (службы управления записями). Время прогона сократилось с 6 часов до 6 минут, а использование ЦП – с 15 минут до 7 секунд, что позволило оперативно составлять прогнозы.
15. **Печать Postscript.** Чтобы получить более четкие диаграммы модели предприятия, мы применили творческий метод «принятия желае-

мого за действительное» и задались вопросом: можно ли графику, созданную на ПК с помощью Freelance, распечатать на новом лазерном Postscript-принтере для VAX? В результате мы разработали способ вывода на лазерный принтер, подключенный к VAX, отчетов, сгенерированных на ПК. Кроме того, характеристики печати стали кодироваться в имени файла, поэтому пользователям не нужно знать в подробностях, как создавать отчет или график. Это средство используется в прогностических отчетах.

16. **Восстановление системы САМ.** В результате сбоя диска наша система автоматизированного управления вышла из строя. Наш процесс производства в значительной мере зависит от этой системы, поэтому нам пришлось искать способ скорейшего восстановления ее работы. Аналитик АСУ и ведущий пользователь предприняли мозговую атаку с целью найти способ автоматического восстановления потерянных данных вместо их повторного ввода вручную. Найденный способ позволил избежать долгого ручного ввода, создать четкий аудиторский след и уменьшить простой автоматической системы управления.
17. **Отчетность системы САМ.** Пытаясь разработать отчеты для отдела управления производством с помощью имеющегося генератора отчетов, мы натолкнулись на существенное ограничение этой программы. Мы прибегли к творческому методу «5W и H» и в результате обошли ограничение на числовые значения в групповых ключах СУБД тем, что «позаимствовали» свободное поле записи для требуемого поля поиска.

Обсуждение полученных успешных результатов подтвердило большую удовлетворенность ими. Участники высказывали такие мнения:

Комментарий 1. «Раньше мне приходилось скрывать, что я работаю над «завиральными» идеями, – я чувствовал, что от меня ждут работы над более практическими решениями. Теперь меня не волнует, если кто-то считает, что я витаю в облаках, – руководство уверено, что мы умеем генерировать хорошие идеи».

Комментарий 2. «На наших совещаниях многие остерегались выступать с «безумными идеями», опасаясь насмешек. Теперь нас не смущают «безумные идеи». Они иногда вызывают смех, но мы знаем, что это, скорее, игривое настроение, а не критика. На насмешки мы спокойно отвечаем, что просто пытаемся проявить творческий подход».

Комментарий 3. «Планируя совещание с пользователями, я задумываюсь над тем, какие творческие методы могут оказаться уместными в той или иной части повестки дня».

Комментарий 4. «Я еще не совсем удобно себя чувствую, предлагая воспользоваться этими творческими методами тем, кто не прошел семинар по творчеству. Но если моя группа заходит в тупик и мы не можем найти решение, я предлагаю какой-нибудь из этих методов, чтобы выбраться из тупика и двигаться дальше».

Комментарий 5. «Для меня одним из неожиданных последствий стало более снисходительное отношение к тем пользователям, которых я всегда считал неспособными к творчеству. Теперь я обычно говорю себе: спокойно, пусть они ищут свой подход к задаче. Они больше способны к творчеству, чем я раньше думал, просто у них менее последовательный подход к поиску решения».

Комментарий 6. «На вопрос о том, нашел бы я творческие решения, перечисленные в нашем списке, не пройдя программу обучения творчеству, я отвечу, что некоторые из них – наверняка. Однако теперь я чаще ищу творческие решения».

Комментарий 7. «В нашей новой творческой среде мы чувствуем себя менее скованными в мыслях. Например, вчера Тереза предложила решение, представляющее собой квантовый скачок в мышлении. И такое в новых условиях бывает с каждым из нас».

Комментарий 8. «Показателем нашего творчества служат усовершенствования в бухгалтерском пакете. Мы придумали и реализовали ряд усовершенствований, которыми удивили экспертов – разработчиков пакета. У них технических специалистов в несколько раз больше, чем у нас, тем не менее, мы смогли внести существенные изменения».

Одной из важных концепций работы проф. Кугера по усилению творческого начала в ИС была необходимость проводить его в восходящем направлении. То есть в то время как большая часть литературы об информационных системах и программировании ориентирует на нисходящие усовершенствования, когда технологии принимаются, изучаются и внедряются с санкции руководства, проф. Кугер решил, что для стимуляции творчества лучшим будет подход «снизу вверх». Его программа обучения была построена на демократичной основе: руководители и технические специалисты учились вместе. По мнению проф. Кугера, это ведет к большей открытости руководства новым идеям и легкости их внедрения по сравнению с традиционным нисходящим подходом. Более того, в данном исследовании, проведенном в UTMC, он обнаружил, что 90% усовершенствований можно реализовать, не получая для этого разрешение у руково-

дства. Это повысило полномочия сотрудников: один из технических работников заявил, что этот период был самым замечательным в его карьере.

Один из руководителей организации, в которой проф. Кугер проводил свое исследование, подвел итог так: «Краткосрочные достижения включают в себя совершенствование творчества в разработке систем, процессов бизнес-систем, приема на работу, эффективности затрат и продуктивности систем. Долгосрочные перспективы выглядят еще привлекательнее. Творчество стало частью нашей корпоративной культуры. Мы стали значительно эффективнее как инициатор перемен в UTMC. И мы уже приступили к повышению эффективности работы компании».

Данное описание применения творческих методов при эксплуатации информационных систем относится к довольно раннему периоду исследований, проводившихся проф. Кугером по этой теме. Позднее, в [Couger 1996], он сумел количественно описать преимущества, полученные UTMC. По его оценке, возврат на инвестиции (ROI) для затрат на обучение творчеству был шестикратным.

Уже говорилось, что в нашей профессиональной области исследования конкретных случаев стимуляции творчества практически отсутствуют. Второй интересный случай, являющийся исключением из этого правила, приводится в [Maiden et al 2004], где описано успешное применение прикладных творческих моделей в формировании требований.

Судя по этому исследованию применения творчества в реальном мире, результаты неплохие. По крайней мере, в этом уголке Колорадо они оказались впечатляющими. Остается только выразить сожаление, что литература о творчестве и регистрация таких конкретных случаев остаются столь редкими в нашей области.

Ссылки

COUGER, MCINTRYE, HIGGINS AND SNOW 1991 – «Using a Bottom-Up Approach to Creativity Improvement in Information Systems Development», *Journal of Systems Management*, Sept., 1991; J. Daniel Couger, Scott C. McIntrye, Lexis F. Higgins, and Terry A. Snow. Excerpts used with permission.

COUGER 1996 – «Creativity & Innovation in Information Systems Organizations», Boyd & Fraser Publishing Co., 1996; J. Daniel Couger.

MAIDEN ET AL 2004 – «Provoking Creativity: Imagine What Your Requirements Could be Like», *IEEE Software*, Sept.,2004; Neal Maiden, Alexis Gizikis, and Suzanne Robertson

Глава 12 | Исторические даты творчества в программировании

Введение
12.1. Первая историческая дата
12.2. Следующие «серебряные пули»
12.3. Творчество и стремление к упрощенности

Введение

Думаю, главное качество творческих людей –
их вечная неудовлетворенность тем, что есть.

Кен Хига, маркетолог
Lord, Aeck, and Sargent, Inc.

«The Creative Edge»,
Atlanta Business Chronicle, Sept. 16, 2005

Есть вид деятельности, важность которой неуклонно растет; это обработка информации, и, управляя ею, лучше не навязывать предварительный детальный список задач, даже если вы можете это сделать.

Билл Уолтон

в специальном выпуске *Cutter IT Journal,*
посвященном роли стратегии, планирования и финансирования
в организации ускоренного программирования, февраль 2006 г.

У нас в обществе любят отмечать исторические даты. Например, 200-летие автомобильной антенны или 100-летие автомобильной подножки. Интересно оглянуться и посмотреть, о чем мечтали люди в те далекие годы.

Много дат можно отметить и в истории *аппаратного обеспечения* компьютеров. В прошлом десятилетии отмечались юбилеи таких древних машин, как ENIAC и EDSAC, и ряда других. Назовите любую машину начала компьютерной эры и можете не сомневаться, что мы отпраздновали ее юбилей и значение в истории. Судя по календарю, скорее всего, это 50-летие.

Но обратите внимание: для *программного обеспечения* исторических дат почти нет. Дело не в том, что очень хочется отметить день рождения перфокарты, блок-схемы или наборного поля. Но я считаю, что в истории программирования были события, достойные праздника. Им и посвящена эта глава. Я излагаю свои представления о важных вехах. Мне также было бы интересно узнать ваше мнение по этому вопросу.

12.1. Первая историческая дата

Прежде всего, можно было бы отпраздновать появление первой компьютерной программы. Но тут история не дает однозначного ответа. Например, некоторые считают, что Ада Лавлейс, в честь которой назван язык программирования Ada, была первым программистом, потому что благодаря ей разностная машина Бэббиджа смогла делать то, для чего она предназначалась. Но в те далекие годы (примерно 150 лет назад) программирование было совершенно иным, и, боюсь, это не то событие, которое большинство из нас с готовностью согласилось бы отпраздновать.

То же касается всех ранних математических/научных приложений. Вторая мировая война создала плодотворную почву для применения компьютеров и программ, и некоторые программы внесли свой вклад в военные действия. Но обычно они выполняли такую работу, как, скажем, расчет таблиц данных – достойное занятие, но едва ли заслуживающее звания эпохального события.

И вот, пока я рылся в пыльных экспонатах, скопившихся в моей памяти почти за 60 лет работы, возникла идея. Наиболее значительной вехой в истории программирования могла быть разработка первого бизнес-приложения. Программы, которая перерабатывала бизнес-данные и составляла отчеты для менеджмента. Конечно, это весьма обыденная работа, но она очень сильно повлияла на развитие бизнеса, сделав его таким, каким мы его видим теперь. И сегодня большинство программ предназначены именно для этой предметной области.

То, что я скажу дальше, пожалуй, может удивить. Первые приложения для бизнеса были написаны в 1951 году. Я сам удивляюсь еще больше, потому что хорошо помню 1950-е – когда в 1954 году я начал работать в этой индустрии, то считал себя настоящим первопроходцем! Оказывается, я опоздал на этот праздник чуть ли не на три года.

Кто же настоящие пионеры этой области? Всем, кто знаком с историей компьютера, известно, что это была английская компания, поскольку Англия сыграла большую роль в начале эпохи компьютеров. Но по-настоящему удивительно, что эта английская компания под названием J. Lyons занималась чаем, мороженым и кондитерскими изделиями (Англия пребудет вечно!).

Самое поразительное в компьютерном новаторстве J. Lyons то, что там не только написали первое бизнес-приложение, но и разработали собственное аппаратное обеспечение для этого приложения! Это была бизнес-версия компьютера EDSAC, разработанного в английском кембриджском университете, позже получившая название LEO.

Вы, наверное, подумали, что бизнес-приложение J. Lyons урожая 1951 года было весьма примитивным. Здесь вас ждет еще один сюрприз! Программы, которые писали в этой небольшой английской компании, вовсе не были наивной стариной. Конечно, там была обработка данных деловых операций, которую вы должны себе представлять, если когда-либо писали программы для деловой сферы. Продажи, склад и все такое прочее – элементы приличного бизнес-приложения.

И это еще не все. Результаты транзакций сравнивались со стандартами, прогнозами и бюджетами. Выявлялись отклонения, и с ними работали специальные менеджеры нижнего уровня. Программное обеспечение позволяло исследовать вопросы «что, если...», задаваемые высшим руководством. В итоге получилась работающая в реальном времени система обработки деловой информации и поддержки принятия решений, которой многие компании позавидовали бы даже сегодня! То, что было создано Lyons многие годы назад, можно сравнить с «анализом и перестройкой бизнес-процессов», куда относится очень многое из того, что мы сегодня называем *«авангардом»*. Вот что сейчас можно прочитать о системе Lyons: «Ирония в том, что в сегодняшних словопрениях в качестве направления дальнейшего развития приводятся заповеди, выработанные свыше 50 лет назад в J. Lyons».

Об этих ранних годах программной отрасли мне поведал Фрэнк Лэнд (Frank Land) из Лондонской школы экономики, принимавший участие в разработках Lyons. Он не раз рассказывал о своей работе в Lyons, но больше всего мне нравится статья, которую он написал для моей книги [Glass 1998], где были собраны воспоминания пионеров программирования, в том числе Лэнда и мои собственные. (Более энциклопедический взгляд на начало компьютерной эпохи с точки зрения ее зачинателя изложен в [Lee 1995]).

Назвав событие, которое считаю наиболее важным в истории прикладных программ, в частности бизнес-приложений, хочу выразить одно сожаление.

Оно заключается в следующем: нам, программистам, следовало бы шумно отпраздновать этот 50-летний юбилей. Но он случился в 2001 году, несколько лет назад.

Мы упустили возможность ввести в календарь еще одно культовое событие. Нехорошо. Разве в эпоху компьютеров есть что-то важнее, чем ознаменование роли программного обеспечения в наступлении этой эпохи?!

Впрочем, можно отпраздновать 100-летие – в 2051 году. Не возьмете ли вы на себя обязанность напомнить об этом миру, когда подойдет время?

Ссылки

GLASS 1998 – «In the Beginning: Recollections of Software Pioneers», IEEE Computer Society Press, 1998; Robert L. Glass.

LEE 1995 – «Computer Pioneers», IEEE Computer Society Press, 1995; J.A.N. Lee.

12.2. Следующие «серебряные пули»

В предыдущем разделе я рассказал о первом примере творчества в истории программирования – о разработке бизнес-приложения и специализированного компьютера LEO в английской компании J. Lyons в 1951 году. Я назвал это поразительным явлением в истории программирования, заслуживающим юбилейных торжеств.

Деятельность J. Lyons была лишь началом длинной цепочки творческих вех в истории компьютеров и программ. Хочется вспомнить и другие события этой цепочки.

Стоит ли рыться в хронологической пыли, выясняя, как программирование дошло до его нынешнего состояния? В конце концов, программирование интересно тем, что мы умеем сегодня, а не (относительно) слабыми достижениями прошлого.

Но послушайте. В нашей области многое успело забыться, и мы то и дело слышим: «Я и не подозревал!» Одни, обсуждая компьютерную историю, допускают ошибки. А другие потом ссылаются на ошибочные заявления, тем самым усугубляя ошибку.

Есть и другой фактор, помимо ошибочности утверждений. Недавно я писал статью, посвященную вечным для меня идеям, таким как структурированные методологии и CASE-средства, и мой молодой коллега указал мне на то, что многие из сегодняшних читателей могут не понять, о чем идет речь, и, что еще хуже, всякое упоминание о них делает мою статью устаревшей. Пришлось согласиться с ним. События 1970–1980-х, вроде структурных методов и CASE-средств, мало интересуют сегодняшних программистов, которые тогда только родились.

Но в то же время мне хотелось возразить. Это часть нашей истории, о которой не следует забывать. Не только из-за полученных нами важных уроков, хотя и их трудно переоценить. Дело в том, что многое из того, что мы сегодня думаем, что знаем, можно прямо (или косвенно) проследить до этих исторических вех. Они составляют богатую историю и материал, на основе которого мы и дальше будем развивать программирование.

Поэтому давайте совершим короткую историческую экскурсию по творческим вехам программирования:

Прежде всего, во времена J. Lyons компьютеры поставлялись пустыми, и писать для них программы нужно было на машинном языке. А машинный язык был творением инженеров, чертовски полезным (способным на любые компьютерные трюки), но явно недружественным для пользователя. (Представьте, что все операции и адреса данных записываются числами, а сами эти числа даже не десятичные, а двоичные, восьмеричные, шестнадцатеричные или еще в какой-нибудь системе из применявшихся в тогдашних компьютерах.)

К счастью, та эпоха довольно быстро закончилась. Сначала появился *ассемблер*, позволяющий кодировать операции и адреса не числами, а символами. С точки зрения исторического значения, это событие не было очень яркой вехой, поскольку не сильно развило возможности программирования. Но с точки зрения творчества, появление ассемблера было шагом к компьютерной обработке символов, а не чисел. А на этом основано большинство других вех в программировании.

Еще одна из фундаментальных идей в программировании появилась примерно одновременно с ассемблером. Программы стали строить с помощью *модулей* – отдельных программных блоков для решения конкретных задач, которые можно было вызывать из основной программы. Эта идея, возникшая в середине 1950-х, и сегодня остается одним из главных достижений в истории программирования. Конечно, ее развивали последующие поколения программистов-практиков, и лет через десять ею занялись теоретики, но мысль о том, что программы можно строить из (многократно используемых) отдельных частей, многие до сих пор находят восхитительной (полагая, что сами до нее додумались!).

Довольно скоро после появления ассемблера и модульного программирования произошли два важнейших события в истории программирования: появились *языки программирования высокого уровня* и *операционные системы*. То и другое случилось в середине или конце 1950-х, и, честно говоря, не могу припомнить, что появилось раньше в тех программирующих организациях, где я работал. Пожалуй, за всю историю программирования это были самые заметные перевороты, совершенные инструментами. Инструмент под названием «компилятор» преобразовывал язык высокого уровня в машинный язык. Первым таким языком был Fortran, который (в несколько измененном виде) применяется по сей день. Вслед за ним появился COBOL, также применяемый в настоящее время. Эти языки были ориентированы на конкретные предметные области: Fortran говорил на языке ученых и инженеров, а COBOL – на языке бизнес-аналитиков.

Операционные системы, как мы уже знаем, служат тем средством, благодаря которому программисты, создающие приложения, могут забыть об утомительной организации взаимодействия с аппаратурой, сконцентри-

ровав свое внимание на той прикладной задаче, которую нужно решить. В общих чертах, операционные системы представляли собой унифицированный набор программных модулей для поддержки сервисных функций аппаратуры конкретного компьютера. В те времена мы так радовались появлению операционной системы, не предполагая, что в какой-то момент (то есть, сегодня) операционные системы станут предметом споров и даже политических разногласий!

1950-е сменились 1960-ми, и эти творческие прорывы упрочили свои позиции в программировании. Возникали новые языки высокого уровня. Появлялись новые и лучшие операционные системы. Скорость этих изменений была поразительной, но они были скорее эволюционными, чем революционными. Эта эволюция включала создание новых языков программирования, независимых от предметной области (например, Fortran и COBOL объединились в PL/1), и даже создание аппаратного обеспечения, также охватывающего разные предметные области (самые первые компьютеры были ориентированы либо на научные расчеты, либо на бизнес). Разумеется, новые операционные системы поддерживали все более разнообразную аппаратуру.

С концом 1960-х связано рождение *академических компьютерных дисциплин*. Сначала появились компьютерные науки (Computer Science), а вслед за ними – информационные системы (Information Systems). Программный инжиниринг (Software Engineering) как отдельная дисциплина появился лет на десять лет позже. Видимо, большинство читателей будет удивлено таким поздним возникновением этих дисциплин в процессе эволюции программирования. (Заметим, что со времени событий в J. Lyons прошло 15 лет.) Множество программ было написано к тому моменту, как за увитыми плющем стенами задумались над тем, как это делать правильно!

Примерно в те же годы разрабатывались некоторые из наиболее основательных *прикладных систем*. Были попытки (в основном, неудачные) создания сложных интегрированных автоматизированных систем управления (АСУ – Management Information Systems, MIS). Примерно тогда же коммерческие производители, такие как SAP и PeopleSoft, начали разрабатывать аналогичные системы, десятилетие спустя достигнув благодаря им громкого и сказочного успеха. Успешно вводились в эксплуатацию системы заказа авиабилетов. Операционная система для IBM 360 была тогда одним из крупнейших приложений и хорошо справлялась со своими функциями. Были разработаны гигантские и невероятно сложные системы для космоса и метеорологии. В далекие 1950–1960-е мы думали, что строим сложные и внушительные приложения. Никто и представить не мог, что ждало нас впереди!

По мере роста объема и сложности приложений в центре внимания оказываются системы, инструменты и идеи, позволяющие бороться с этой сложностью. Сначала в 1970-х появилось *структурное программирование*. Оно представляло собой ряд методологических рекомендаций по поводу того, как надо и не надо писать «структурированные» программы. В структурном программировании было два примечательных аспекта. Первый – его восхваление как революции в программировании, в связи с чем оно было принято на вооружение практически всеми программистами почти всех организаций. Второй – отсутствие каких-либо исследований, подтверждающих существование столь превозносимых достоинств. Десятью годами позже изучение литературы, оценивающей эффективность структурного программирования, дало двусмысленные результаты, никоим образом не подкреплявшие первоначальные претензии. Важно заметить, что несмотря на утихшую шумиху, сегодня большинство программистов уверены в полезности структурных методов, и большинство современных программ в некотором смысле «структурированы».

Тот случай неоправданного восхваления отнюдь не последний в отрасли. *Инструменты и языки* были представлены как методы «автоматизации» программирования, благодаря которым эту работу осилит каждый – не обязательно профессиональный программист. В 1980-е заявляли, что средства CASE (автоматизации программирования) и 4GL (языки четвертого поколения) сделают это возможным. Неважно, что многие CASE-средства были куплены, а затем брошены и забыты (такие программы в шутку называли «полочными»). Неважно, что практики и теоретики совершенно разошлись во взглядах на 4GL (практики считали их языками генерации отчетов по базам данных; теоретики – непроцедурными языками, в которых программист описывает, какие действия нужно выполнить, но не их порядок). Неслучайно примерно в это время Фред Брукс написал свою историческую статью «Серебряной пули нет», в которой высказал мнение, что все революционные прорывы в программировании уже свершились и, по всей вероятности, в будущем нас не ждет ничего столь же выдающегося, как языки и операционные системы 1960-х.

Что произошло с CASE и 4GL потом? Подозреваю, что мы до сих пор их применяем, но сами термины пользуются таким неуважением, что их редко встретишь. И уж, конечно, широко разрекламированные успехи так никогда и не были достигнуты.

Структурное программирование не было последним словом в представлениях о революционной роли методологии. Последним криком моды стало *объектно-ориентированное программирование* (*ООП*), от которого ждали чудес, как прежде от структурного. Сегодня трудно судить об ООП непредвзято, поскольку пока еще не предложена очередная революционная технология, готовая его сменить. Есть разные данные по поводу широты

применения ООП (некоторые утверждают, что он распространен повсеместно, но те исследования, с которыми я ознакомился, указывают, что им пользуется меньше 50% компьютерных организаций). Относительно преимуществ ООП мнения также расходятся. Предполагается, что его методы облегчают работу новичкам, однако исследования показали, что новички лучше справляются с функциональным, чем с объектно-ориентированным подходом. Считается, что ООП должно способствовать повторному использованию кода (помните наш разговор о модульном программировании?), и есть исследования, показавшие, что в некоторых предметных областях добрые 70% кода можно использовать повторно, а не писать заново (хотя другие исследования показали почти такие же результаты для методов, не являющихся объектно-ориентированными). ООП должно также позволить создавать программный код в объектах решения, непосредственно взаимосвязанный с объектами решаемой задачи, но сейчас в нем приняты «сценарии использования» (use cases), а это явно функционально-ориентированный подход к определению системных требований. И снова: по мере снижения ажиотажа вокруг ООП становится очевидно, что это хороший, но не идеальный способ разработки программ.

Что еще осталось из творческих вех в истории программирования? Вклад последнего времени – гибкое программирование (Agile) и ПО с открытым исходным кодом (Open Source). У каждой из этих технологий найдутся сторонники, считающие их великолепными творческими достижениями в области разработки программ. И другие – те, кто считает Agile-методы пригодными лишь для небольших приложений некоторых типов, а Open Source – эмоциональным всплеском, более интересным своим фанатическим поклонникам, чем миру программирования в целом.

12.3. Творчество и стремление к упрощенности

Несомненно, все исторические творческие вехи в программировании, перечисленные в предыдущем разделе, были действительно творческими. Но интересно: о чьем именно творчестве мы говорим?

Эта книга посвящена, в основном, творчеству в программном инжиниринге. Больше всего меня волнуют проблемы, решение которых поможет создателям программного обеспечения делать свое дело лучше и более творчески. Но все ли отмеченные вехи этому способствовали?

Этот важный и интересный вопрос несколько десятилетий назад имел яркую политическую окраску. Но прежде чем говорить о том, какими печатями отмечены эти вехи, обратимся собственно к вопросу.

Первый ответ, который пришел мне в голову, это «разумеется, все эти события послужили развитию творчества программистов», – думаю, вы ав-

томатически ответите так же. В конце концов, разве все эти языки высокого уровня, различные методологии и инструменты программирования не внесли внушительный вклад в наше умение выполнять свою работу?

Но после некоторого размышления у вас могут возникнуть некоторые сомнения, как они возникли у меня. Да, все эти нововведения очень помогли программистам. Но способствовали ли они развитию творчества?

Прежде чем браться за этот вопрос, нужно объяснить, где происходил первичный творческий акт. Очевидным образом, творчество, связанное с этими вехами, происходило в головах тех, кто их породил. Представьте первого человека, который изобрел язык высокого уровня (ЯВУ) и компилятор как необходимый для этого инструмент. Чтобы придумать ЯВУ+компилятор, необходимо было увидеть «компьютер» в совершенно ином свете, как обработчик символьной информации. На самом деле, примерно в это время термин «компьютер» (вычислительная машина) начал устаревать, потому что применение этой машины перестало ограничиваться арифметическими действиями над числами и распространилось на сложные логические операции над нечисловыми символами. Комбинация ЯВУ+компилятор стала революцией – «серебряной пулей», затронувшей, огромное число уровней.

И все же – усилило ли это изобретение творческие возможности программистов? Глубоко поразмыслив, я по-прежнему отвечаю на этот вопрос утвердительно. С учетом легкости написания программы на ЯВУ по сравнению с ассемблером программист освободился от деталей, относящихся к конкретной машине, и смог направить свой ум исключительно на решение прикладной задачи. А если это не усилитель творчества, то я не знаю, что вам еще нужно. Правда, это иной вид творчества, чем у создателя оригинальной идеи.

Надо сказать, что почти все первые серебряные пули явно стимулировали творчество программистов, по той же логике, что я привел для ЯВУ+компилятор. Операционные системы давали программисту возможность сконцентрироваться на задаче, не вдаваясь в машинную архитектуру. Модульное программирование позволило программисту работать с компактными понятиями, вместо свободно (порой самовольно) распространяющихся на все пространство задачи/решения.

Спор по поводу творчества начинает проясняться, когда мы переходим к теме методологий. Дают ли методологии вроде структурного или объектно-ориентированного программирования больше возможностей для творчества или они лишь предоставляют среду, в которой программисту труднее совершать ошибки? Можно найти аргументы в пользу обеих сторон, но в данном случае мне меньше хочется быть «за», чем в случае ЯВУ+компилятор. Думаю, поддерживая применение методологий, руко-

водители – тогда и сейчас – стремились к ужесточению контроля над процессом программирования. А если так, то методологии вообще не имеют отношения к творчеству.

Теперь мы готовы к обсуждению политического аспекта, обещанному выше. В конце 1970-х проблема значения крупных исторических событий в истории программирования вызвала довольно острую полемику. Два автора, политически скорее левацкого толка, высказали мнение, что все крупные исторические события в программировании, о которых мы говорили, в целом представляют собой хитрость руководства, заговор с целью не повысить творческий уровень своих работников, а «деквалифицировать» их! А зачем руководителям деквалифицировать своих рабочих? Чтобы было проще управлять ими и контролировать создаваемые ими продукты.

Вот эти два автора, которые тогда вышли на компьютерную сцену, чтобы смешать политические и компьютерные понятия:

- Филип Крафт (Philip Kraft) [Kraft 1977], социолог, рассматривавший компьютерную область преимущественно с точки зрения трудовых отношений («Я не скрываю, – пишет он в указанной книге, – что стал ярым сторонником партии программистов»).
- Джоан Гринбаум (Joan Greenbaum) [Greenbaum 1979], бывшая программистка, исповедовавшая радикальные взгляды на общественные системы и часто цитировавшая Карла Маркса в подкрепление своих высказываний.

А идея, излагавшаяся в этих двух книгах, заключалась в том, что все эти важные вехи, о которых мы говорили выше, особенно методологические, имели целью сделать работу программистов рутинной и «деквалифицировать» их.

Какие же аргументы они приводили?

Прежде всего, оба автора в своих книгах опирались на исследование, в котором проводился опрос программистов и их начальников по поводу происходящего в мире программирования. Крафт усмотрел в результатах этих опросов ряд современных технических тенденций, которые он счел преднамеренными попытками начальников понизить квалификацию работников: структурное программирование, модульность и готовые программы, а также бригады под руководством главного программиста (метод организации, в чем-то сходный с медицинским, когда в хирургической бригаде есть один главный хирург). В каждом из этих случаев, по мнению Крафта, вместо роста производительности труда (который отмечали почти все работники отрасли) происходило усиление контроля руководства над процессом программирования (например, сокращение логических структур, разрешенных к использованию программистами, ес-

ли это было структурное программирование; максимизация объема готового кода, используемого вместо написания новых программ, если речь шла о модульности; бригадная организация, которую он считал самым структурированным, иерархизированным и сословным типом организации из всех существующих в промышленном мире).

Любопытно, что Крафт также рассматривал закрытость вычислительных залов (в те времена программистов не подпускали к компьютерам – на них работали профессиональные операторы) как очередную попытку администрации деквалифицировать своих работников – на этот раз, путем разделения «умственного» и «физического» труда.

Поскольку сегодня компьютеры везде, даже если закрытые машинные залы действительно были попыткой руководства усилить контроль, то она явно закончилась полным провалом.

Крафт мельком отметил, что сами создатели средств повышения производительности не стремились деквалифицировать программистов, но своей позиции при этом не изменил.

Гринбаум в свой книге (позже) разделила точку зрения Крафта на структурное программирование, но обошла вниманием другие его тревоги вроде модульности и бригадной организации. Однако она тоже уверяла, что так называемые средства повышения продуктивности в действительности представляют заговор руководителей с целью усилить контроль программистов «под видом эффективности», как гласило название книги.

В дополнение к разоблачению заговора руководителей с целью деквалификации своих работников Крафт и Гринбаум с удивлением обнаружили, что программисты, хоть и жаловались на руководителей, вовсе не пытались сопротивляться или возражать им. Например, Крафт писал, что он был «поражен тем, что программисты регулярно принимали точку зрения своего руководителя, даже если она была очень далека от точного описания реального положения программиста».

Оба автора отметили тенденцию к утрате увлекательности процесса программирования (тема, к которой мы уже обращались в этой книге) и усилению контроля со стороны руководства – по их словам, опросы программистов систематически выявляли эту тенденцию. Гринбаум отметила, что «в 1960-е программистами были люди бунтарского склада – умные и творческие первопроходцы; сегодня они другие – технически подготовленные и усидчивые». Крафт во многом вторит ей: «Еще не выросло второе поколение программистов, как программирование перестало быть сложной работой для творческих людей. Разделенное на части, упрощенное, оно стало массовым производством». (Обратите внимание: здесь снова проявляется тревога о творчестве. Крафт и Гринбаум явно считали творческое начало необходимым для программирования).

Конечно, легко судить задним числом, и сейчас мы понимаем, что большинство предсказаний Крафта и Гринбаум не сбылись. (Не стану хвастать, но кое-кто из нас понял это сразу!)

- Структурное программирование – «средоточие усилий руководства по деквалификации программистов» – в целом исчезло с горизонта, во всяком случае, теоретически, уступив место объектно-ориентированному программированию, которое никто не называет «деквалифицирующим».
- «Мало сомнений в том, что процесс фрагментации работы и деквалификации программистов будет продолжаться и усиливаться» – это утверждение выглядит смешным в сегодняшнем мире все более сложных программных систем.
- «Бюрократические формы управления особенно легко применить для регламентации и контроля сопровождения программного обеспечения» – это заявление было неверным тогда и остается неверным сегодня, поскольку сопровождение по-прежнему не укрощено.

Несмотря на все заблуждения Крафта и Гринбаум, нужно признать, что они подняли ряд интересных и важных вопросов, сделав это достаточно рано – больше тридцати лет назад. Любые попытки деквалифицировать программистов, несомненно, контрпродуктивны (и вредят творчеству) в мире программирования, каким мы его знаем сегодня (хотя едва ли кто-то может гарантировать, что руководители неспособны на такую уловку). Но их основная идея о том, что все творческие достижения в программировании в действительности были нацелены на деквалификацию и против творчества, довольно легко может быть опровергнута: достаточно лишь взглянуть, насколько сложной и по сей день остается задача создания программного обеспечения.

Однако политическую проблему, поднятую Крафтом и Гринбаум, стоит запомнить. Если когда-либо руководители задумают сделать процесс программирования неквалифицированным, все творческие технические специалисты должны быть готовы дать им отпор.

Ссылки

GREENBAUM 1979 – «In the Name of Efficiency», Temple University Press, 1979; Joan M. Greenbaum.

KRAFT 1977 – «Programmers and Managers», Springer-Verlag, 1977; Philip Kraft.

Часть III

Краткий обзор творчества в других областях

Введение
13. Организационное творчество
14. Творческая личность
15. Помощь компьютера в творчестве
16. Парадоксы творчества
17. Так было всегда

Введение

Те, кто исследовал творческий процесс, давно выделили два типа мышления: анализ и синтез. Иногда латинскому слову cognito, означающему «я думаю» в смысле анализа или разбора по частям, противопоставляется intelligo, означающее «я понимаю» в смысле проникновения в природу явления... Глядя на мир, мы можем увидеть, чем вещи разнятся (найти различие) или в чем они схожи (найти аналогию). В первом случае мы обычно получаем новые категории, а во втором обычно меняем контекст.

Эллен Дж. Ланджер

«Mindfulness», Addison-Wesley, 1989,
печатается с разрешения
Addison-Wesley Publishing Company, Inc.

Есть что-то завораживающее в программировании. Мы отдаем ему свою энергию, свой талант, свои интересы. Можно всю жизнь заниматься вещами, связанными с программированием, и быть вполне счастливым.

Но тот, кто направляет свое внимание в одном единственном направлении, через какое-то время начинает понимать, что его кругозор сужается, а рассудительность страдает. Подвержены ли подобному недугу те из нас, кто слишком долго и старательно концентрировался на программировании?

Думаю, да. В настоящей, третьей, части книги дан краткий обзор творчества в других областях. Из него можно извлечь некоторые полезные уроки.

Особого внимания заслуживает такая древняя область, как решение задач. Ее история необычна. Хотя решению задач столько же лет, сколько человечеству, и о нем писали на протяжении веков, пришествие компьютеров вызвало новый интерес к этой сфере. В начале 1960-х в области решения задач был проведен ряд примечательных исследований.

Среди этого нового всплеска интереса к решению задач особый интерес представляют следующие три результата:

1. То, что мы называем жизненным циклом программ, в действительности представляет собой универсальный метод решения задач (эта тема разрабатывается ниже в главе 17).
2. По мере роста сложности задач все большее значение приобретают эвристические методы, а формальные методы начинают давать сбой (эта тема разрабатывалась в части I).
3. Общие методы решения задач, позволившие добиться больших успехов в области компьютерных вычислений к настоящему времени, считаются «слабыми» методами решения задач, а «сильными» считаются специализированные методы (наглядный пример: отворачивание гайки с помощью разводного ключа или ключа на конкретный размер).

Будучи взятыми по отдельности или вместе, эти три результата предполагают резко различающиеся между собой подходы к практике и исследованиям в области программирования в будущем. Например, в будущем возможны: какие-то изменения в жизненном цикле вместо полного отказа от него; переориентация с формальных методов на эвристические, что существенно противоречит сегодняшним предпочтениям в мире программирования; переход от универсальных общих методов решения к узкоспециализированным – например, к созданию специальных методологий, инструментов, языков или даже компьютеров.

К чему я клоню? К тому, что выйдя за пределы своей области, мы можем увидеть ее с другой точки зрения, дающей гораздо лучшее представление. При этом можно обнаружить фундаментальные проблемы, связанные с нынешними направлениями развития. Это и составляет цель данной части книги.

В следующих двух главах мы рассмотрим творчество с точки зрения организации и личных качеств. (Легко представить, что в нынешнем тысячелетии для решения все усложняющихся задач, с которыми мы встречаемся в своей отрасли, потребуются как личные способности, так и новые типы организации.)

Затем мы рассмотрим вопрос применения компьютеров для расширения творческих возможностей человека в других областях (лучший способ расширить кругозор!). На сегодня результаты экспериментов в этой области вызывают удивление – и разочарование.

Возможно, это разочарование частично связано с фундаментальными парадоксами в области творчества. Эти парадоксы обсуждаются в следующей главе.

Наконец, дополнительную перспективу этой части книги, которая и так занята перспективами, придаст повторное рассмотрение темы книги с точки зрения связи между нашими проблемами и опытом, накопленным в других областях. Один из авторов пишет о «классическом противоречии между управлением и творчеством».

«Так было всегда», – говорим мы в главе 17.

Главная идея данной книги – что в программировании должны сочетаться административные методы и стимулирование творчества – отнюдь не нова. И это, пожалуй, главный урок, который можно извлечь из краткого обзора творчества в других областях.

Глава 13 | Организационное творчество

В новом тысячелетии потребуются смелые творческие мужчины и женщины, способные претворить свои мечты в реальность.

Др. Кирпал Сингх, профессор
Сингапурского университета менеджмента,
из приветствия семинарам по творчеству

Очень часто, когда перед нами встает задача, мы не созываем совещание, а сообщаем, в чем она состоит, например, с помощью электронной почты, после чего каждый может размышлять над этой задачей в удобное для него время.

Лори Энн Голдман, CEO, SPANX, Inc.

«The Creative Edge», *Atlanta Business Chronicle*,
16 сентября 2005 г.

Вопрос о творчестве возник давно. Это значит, что по данной теме может быть много результатов исследований в других областях, верно?

Сколь ни удивительно, но это не так. Согласно обстоятельному исследованию проблем творчества в более широком пространстве (не только в программировании) есть «ограниченное количество исследований творчества в организациях», зато «исследования организационного новаторства проводились на протяжении десятилетий» [Woodman, Sawyer, and Griffin

1993]. Далее в статье развивается эта тема: «ученые до сих пор поразительно мало знают о том, как происходит процесс творчества» и «с практической стороны, мы также мало знаем о том, как успешно развивать и направлять личное и коллективное творчество в организациях».

Частично проблема связана, по мнению авторов, с довольно слабой работой по «выделению составляющей творчества из более общей концепции новаторства», проделанной исследователями организаций. В чем они видят различие между творчеством и новаторством? «Мы определяем организационное творчество как подмножество более широкой области новаторства. Новаторство в этом случае рассматривается как подмножество еще более широкого контекста организационных изменений... Творчество может породить новый продукт, услугу, идею или процесс, которые реализуются посредством новаторства, но и адаптация ранее существующих продуктов или процессов – тоже новаторство». Иными словами, творчество задумывает что-то новое, а новаторство применяет творческие и/или модифицированные идеи на практике.

Но отступим немного назад и посмотрим, какая картина обрисована в [Woodman, Sawyer, and Griffin 1993]. Предмет, который их заинтересовал и был положен в основу статьи, назван ими «организационным творчеством». Организационное творчество происходит, по их словам, в результате взаимодействия творчества индивида и группы. «Организационное творчество является функцией творческих выходов составляющих его групп и контекстных влияний (корпоративной культуры, систем вознаграждения, ограничений, накладываемых на ресурсы, более широкого окружения вне системы и так далее)... Групповое творчество является функцией результатов индивидуальных творческих усилий, свойств группы, процессов группы и контекстных влияний». Иными словами, хотя индивидуальное творчество играет растущую и важную роль в организационном творчестве, есть и другие значимые факторы.

Поскольку основным конструктивным элементом является индивидуальное творчество, авторы рассматривают его несколько глубже. Им представляется, что его образуют как особенности личности, так и когнитивные факторы. Вот конкретные особенности личности, которые они выделяют:

- Высокая оценка эстетических факторов в работе
- Широкие интересы
- Приверженность к сложности
- Энергичность
- Независимость суждений
- Автономность

- Интуиция
- Уверенность в себе
- Способность справляться с противоречивыми чертами в самооценке
- Прочное ощущение себя как творца
- Настойчивость
- Любознательность
- Интеллектуальная честность
- Когнитивные факторы
- Легкость ассоциативной деятельности
- Легкость в выражении
- Легкость метафорической деятельности
- Легкость в образовании понятий
- Легкость речи
- Легкость в выборе слов
- Легкость в практическом восприятии идей
- Оригинальность

Такой подробный список творческих черт появился в результате любопытного исследования творческой деятельности. Авторы пишут, что «для первых исследований творчества часто был характерен сбор биографических и исторических сведений об известных творцах». Однако анализ этих сведений был не слишком плодотворным: «Попытки эмпирического сопоставления этих факторов привели к такой факторной сложности, что теоретическая интерпретация стала практически невозможной». Чтобы справиться с этой проблемой, а также заполнить очевидные пробелы в биографическом методе, другие исследователи стали подключать взаимодействие личностных данных с биографическими. Согласно авторам статьи некоторые исследователи «продемонстрировали, что качества личности в совокупности с биографическими данными позволяют предсказать творческое поведение».

Главное достоинство этой статьи, помимо солидного перечня ссылок на более раннюю литературу о творчестве, заключается в формулировании ряда представлений о творчестве в виде гипотез. К сожалению, статья ничего не говорит о проверке этих гипотез, оставляя ее в качестве упражнения для будущих исследователей. Вот эти гипотезы.

Относительно индивидуального творческого показателя:

1a: Растет, если групповые нормы поддерживают свободное распространение информации.

1b: Снижается, если групповые нормы формируют высокий уровень подчинения.

1c: Растет, если корпоративная культура поощряет рискованное поведение.

1d: Снижается, если система поощрения строго учитывает творческие результаты, жестко привязывая их к внешнему поощрению.

Относительно группового творческого показателя:

2a: Растет благодаря внутригрупповому разнообразию (diversity).

2b: Снижается при деспотическом стиле руководства.

2c: Нелинейно взаимосвязано со слаженностью группы

2d: Растет благодаря структурам и культурам с высоким уровнем сотрудничества.

Относительно творческого показателя организации:

3a: Растет при наличии легкодоступных ресурсов.

3b: Снижается при наложении ограничений на потоки информации и каналы связи внутри системы.

3c: Растет при наличии естественных структур (таких, как матрица, сеть, сотрудничающие группы).

3d: Снижается при ограничении обмена информацией внутри окружения.

Многие из этих гипотез, оценка которых отсутствует в статье, взяты из трудов других ученых, писавших о творчестве. Например, авторы указывают, что «вмешательство в мотивацию в виде систем оценок и поощрений может отрицательно сказаться на внутренней мотивации к творческой задаче, переключая внимание с эвристических аспектов творческой задачи на технические или основанные на правилах аспекты задач». Заметим, что из этого положения прямо следует гипотеза 1d.

Также следует обратить внимание на явное предположение исследователей о том, что творчеству вредят «технические или основанные на правилах аспекты задач» и способствуют «эвристические аспекты». Это несколько оправдывает изложенную в других разделах книги точку зрения, в соответствии с которой формальные, методологические, дисцип-

линарные методы разработки программного обеспечения могут препятствовать творческим процессам, которые тоже нужны.

Вот другие цитируемые авторами результаты, которые могут подкреплять высказанные гипотезы:

> «Количество формальных уровней контроля и количество сотрудников, занятых R&D, отрицательно коррелировали с новаторством [не творчеством], тогда как [малый] размер команд в исследовательских проектах... коррелировал положительно...»
>
> «Мы предполагаем, что творчество должно усиливаться благодаря адаптивным, гибким организационным структурам...»
>
> «Разумно предположить, что когда вознаграждение зависит от *функциональных руководителей*, инженеры боятся, что нестандартное поведение будет оцениваться отрицательно. Однако когда вознаграждение зависит от *руководителей проектов*, оценивается общий результат независимо от способов, примененных для решения задачи».

Интересно, что это возвращает нас к уже обсуждавшемуся в этой книге вопросу о том, что чрезмерное внимание, уделяемое процессу, может помешать творчеству и что ориентация на продукт – конечную цель руководителя проекта – более способствует раскрепощению работников для творчества.

Какие же *собственные* выводы делают авторы статьи? «Чтобы понять творчество в социальном контексте, необходимо изучать творческие процессы, творческие продукты, творческих людей и творческие ситуации. Полезная теория организационного творчества должна предложить систему, достаточно сложную и богатую, чтобы объединить эти четыре компонента».

Хороший зачин для будущих исследований есть. Но эту большую работу еще предстоит выполнить.

Ссылки

WOODMAN, SAWYER AND GRIFFIN 1993 – «Toward a Theory of Organizational Creativity», *Academy of Management Review,* Vol. 18, No. 2, 1993, Richard W. Woodman, John E. Sawyer and Ricky W. Griffin.

Глава 14 | Творческая личность

На прагматике шляпы квадратик,
он в квадратиках комнат, прагматик.
Или так: в равнобедренный он
треугольник-тюрьму заключен.
Если ж больше его взволновала
форма конуса с формой овала,
если им исповедана вера
в форму полулуны, –
пусть заузит на конус штаны
и наденет сомбреро.[1]

Уоллес Стивенс
из «Six Significant Landscapes»

Какую личность можно считать творческой?

Это трудный вопрос. Трудный не только потому, что есть множество исследований, давших разные ответы, но и потому, что не прекращаются жаркие споры о том, существуют ли такие личности вообще.

В центре споров о существовании творческих личностей – вопрос о том, можно ли творчеству научиться. И ответ на него, конечно, очень важен. Исходя из нашей уверенности, что некоторая часть задач программирования требует творчества, вопрос сводится к следующему: «Следует ли искать программистов с врожденными творческими способностями или можно воспитать способность к творчеству у имеющихся работников?»

---

[1] Перевод Владимира Гандельсмана.

Если к творчеству способны лишь немногие, это важный, хотя и досадный, ответ. Прежде всего, это очень осложнило бы подбор программистов для работы, потому что наличие творческих способностей среди прочих умений кандидатов становится критически важным. А досадный он, поскольку означает, что лишь немногие программисты могут справиться с задачами, требующими творчества.

К счастью, большинство специалистов по творчеству считают, что творчеству можно научиться. В особенности это хорошая новость для тех, кто предлагает курсы по развитию творчества, потому что их старания бессмысленны, если творчество – такая вещь, которая у вас либо есть, либо нет. Но это хорошая новость и для всех остальных, потому что тогда не нужно беспокоиться – а вдруг передо мной встанет барьер творчества, который я не смогу преодолеть?

Но оставим пока в покое этот вопрос и займемся другим, с ним связанным. Допустим, что есть люди, намного превосходящие остальных в своих творческих способностях. Мы не говорим, что все остальные к творчеству не способны, просто есть люди, заметно превосходящие остальных в этом деле. Каковы признаки таких творческих людей?

По этому вопросу есть много интересных исследований. Вот некоторые особенно примечательные ответы.

> «Одна из черт, которую я сразу заметил, – пишет Дэвид Джонсон [Johnson 1990] в необычном издании под названием «Midnight Engineering» (журнал для тех, кто подумывает о собственном техническом бизнесе), – их напряженная сосредоточенность на работе. Временами казалось, что они живут в ином мире, почти не обращая внимания на то, что происходит вокруг. Их поведение не назовешь антиобщественным, но они явно предпочитали работать со своими компьютерами или оборудованием, чем с живыми людьми».

Джонсон продолжает: «Когда они не сосредоточены на какой-нибудь проблеме, с ними легко ладить, и у них хорошее чувство юмора. Однако я заметил, что они гораздо хуже других переносят отвлечения. Работая над очень сложной проблемой, они рассматривают громкую музыку, канцелярскую работу, совещания и составление отчетов как вмешательство или отвлечение...

Если они и прерываются, то лишь на чтение. Их жажда знаний неутолима. Неважно, о чем читать, лишь бы было интересно... Возможно, из-за этой широты интересов лишь немногие из тех, с кем я сталкивался, имели законченное высшее образование, а многие часто переходили из одной технической области в другую...

Широта интересов творческих людей, с которыми я был знаком, проявлялась и в том, как они решали задачи. Я заметил, что, решая задачи, они не слишком полагались на математический аппарат. Если требовались расчеты, они проводились очень приближенно. Им было проще спаять схему и проверить свою конструкцию, чем заниматься исследованиями на бумаге...»

Неформальный анализ творческих личностей, сделанный Джонсом, дает нам глубокое практическое понимание. Но нет ли более строгих, научных, исследований по тем же вопросам?

Как и следовало ожидать, есть. [Barron 1958] предлагает нам анализ «психологии воображения» в виде ряда интересных исследований.

Итог исследований Баррона: «Творческие личности легче других справляются со сложностью и видимым беспорядком». Как это выявилось в его исследованиях? Вот особенно интересный ответ:

> «При демонстрации чернильных пятен в тесте Роршаха творческие личности с необычайной настойчивостью предлагают такую интерпретацию, которая учитывает все детали в одном синтетическом образе. Поскольку некоторые пятна расположены весьма хаотично, такая склонность к синтезу увеличивает сложность задачи упорядочения. Она также показывает творческую реакцию на беспорядок, когда возникает стремление найти красивый новый порядок, приносящий больше удовлетворения, чем могла бы сделать более простая конфигурация».

Баррон также обнаружил, что творческие люди не склонны уступать общему мнению группы. Он поставил ряд изобретательных экспериментов, в которых большинство участников договаривались между собой дать неправильный ответ, а тестируемый субъект, зная о том, какой выбор сделало большинство, должен был представить свой ответ. Лишь около 25% тестируемых смогло противостоять ошибочному общему мнению. Баррон дополнительно исследовал этих людей и обнаружил, что свою независимость они выражают так:

> «Я люблю играть с новыми идеями, даже если потом оказывается, что они не стоят потраченного времени».
>
> «Некоторые друзья считают мои идеи неосуществимыми или даже безумными».
>
> «Незавершенное и несовершенное часто привлекает меня больше, чем законченное и безупречное».

В результате этого и других тестов Баррон пришел к следующим выводам относительно творческих людей:

«Они исключительно наблюдательны и ценят точные наблюдения».

«Нередко они высказывают часть правды, но делают это очень ярко; та часть, которую они высказывают, обычно бывает непонятой... и незамеченной».

«Они видят вещи так же, как другие, но в то же время иначе».

«Они, таким образом, независимы в своих познавательных способностях, но при этом ценят ясное понимание. Они не пожалеют трудов для корректного изложения».

«Стимулами для них служат... как стремление к самосохранению, так и интересы человеческой культуры и ее будущее».

«Они от рождения обладают мощным умом; они могут удерживать в нем сразу много идей и сравнивать их между собой – отсюда их способность к более плодотворному синтезу».

«Они от природы энергичны и располагают исключительным запасом психической и физической энергии».

«Обычно они ведут более сложную жизнь, ищут напряжение и получают удовольствие при его разрядке».

«Они чаще, чем большинство других людей, обращаются к подсознанию – фантазии, мечтам, миру воображения».

«У них исключительно широкое и гибкое восприятие самих себя».

«Творческая личность одновременно более примитивна и более культурна, более деструктивна и более конструктивна, более безумная и здоровая, чем средний человек».

Теперь позвольте задать вам необычный вопрос. Читая эти утверждения, анализировали ли вы себя, пробовали ли оценить собственный творческий инстинкт? Лично я это делал. Думаю, все мы чувствуем в себе что-то творческое, некое сродство с неуловимым понятием «творческой личности». Иными словами, можно не считать себя творческим человеком, но знать, что сможешь быть творческим, если понадобится. Это, несомненно, приятное и важное чувство. Вы познакомились с творческой личностью – с собой.

Ссылки

Barron 1958 – «The Psychology of Imagination», *Scientific American*, Sept. 1958; Frank Barron.

Johnson 1990 – «The Creative Person», *Midnight Engineering*, January/February 1990; David Johnson.

Глава 15 | Помощь компьютера в творчестве

Программное окружение может как способствовать, так и... препятствовать получению нужного результата.

Джойс Дж. Елам и Мелисса Мед

«Can Software Influence Creativity?»

Решения, полученные с помощью программ, были оценены как менее творческие.

Дуглас Е. Дьюренд и Сьюзи Вангусс

«Creativity Software and DSS: Cautionary Findings»

Большая часть материала этой книги обращена к внутренним проблемам. Иными словами, она посвящена обсуждению роли творчества в создании программ: внимание обращено внутрь собственно области программирования.

Но в данной главе внимание направлено вовне. Мы обсудим, как компьютеры и программы могут помочь людям проявить способность к творчеству. Дело в том, что внутри компьютерной и программной области есть довольно активное сообщество, интересующееся проблемами творчества за ее рамками.

Например, весной 1993 в Англии состоялся международный симпозиум на тему «Creativity and Cognition» (Творчество и познание). Название не предполагает ничего необычного для этой конференции, зато подзаголовок звучит так:

> «Художники, музыканты, дизайнеры, специалисты по теории познания и компьютерам представят идеи о творчестве и познании на стыке искусства и науки».

Это была настоящая междисциплинарная конференция, на которой обменялись своими мыслями эксперты по аппаратному и программному обеспечению компьютеров. По материалам симпозиума в начале 1994 года издательство MIT Press опубликовало книгу.

На конференции обсуждались следующие темы:

- Концепции, процессы и модели вычислений
- Искусство и наука: пересечения и разграничения
- Компьютерная технология: методы и инструменты
- Размышления о художественной практике; исследования практики дизайна

Взаимодействие разных дисциплин нашло отражение в названиях некоторых представленных докладов: «Towards Artificial Creativity» (На пути к искусственному творчеству), «Using the Computer to Augment Creativity: Computer Choreography» (Усиление творчества с помощью компьютера: компьютерная хореография), «Culture, Knowledge and Creativity: Beyond Computable Numbers» (Культура, знания и творчество: за пределами вычислений) и «Tamed Equations» (Укрощенные уравнения). Особенно восхищает образ компьютерной хореографии с применением укрощенных уравнений!

Все эти междисциплинарные взаимодействия проникнуты главным вопросом: действительно ли применение компьютеров усиливает творческие возможности в разных областях? Казалось бы, ответ должен быть положительным, особенно с точки зрения тех, кто связан с компьютерами и привык к разным компьютерным чудесам. Но есть ли надежные факты, подкрепляющие интуицию?

Есть два исследования на эту тему. Любопытно, что они относятся не к миру искусства, где можно было бы ожидать наибольший интерес к творчеству, а к миру бизнеса. У [Elam 1990] и [Durand 1992] мы находим исследование пользы программных пакетов для стимуляции творчества в области принятия решений.

Обычно системы поддержки принятия решений, СППР (decision-support systems, DSS), ориентированы на конкретную задачу, а методы решения

реализуются с помощью баз данных и моделей, специфичных для этой задачи. Заметим, что обычная СППР в некотором отношении представляет собой полную противоположность тому, чем могла бы быть СППР, стимулирующая творчество, поскольку ориентирована на конкретную задачу и применяет структурированный подход к решению, тогда как творческая СППР предположительно должна быть применима к ряду различных задач и способствовать различным свободным, повышающим творчество навыкам.

Первое исследование

В первом из упомянутых исследований [Elam 1990] авторы взяли коммерческую СППР, усиливающую способность к творчеству, и поставили эксперимент, чтобы проверить ее полезность. Выбранная программа (ods/CONSULTANT производства Organization Development Software) на момент исследования еще не вышла на рынок, но авторы решили, что по своим характеристикам она больше подходит для задач исследования, чем имеющиеся на рынке. (Более подробное обсуждение программных пакетов для стимуляции творчества см. в [Thierauf 1993]). Этот инструмент предлагал стимулирование генерации идей, помощь в мозговой атаке, провоцирование вопросов и группировку фрагментов идей, а также количественную поддержку определения приоритетов/численной оценки идей, выявления взаимозависимостей между идеями и классификации идей по признакам, задаваемым пользователем.

В исследовании приняли участие три группы, которые должны были решить один и тот же набор задач, но использовать при этом три разных подхода:

1. Не пользоваться компьютерами.
2. Использовать ods/CONSULTANT по следующей схеме (вариант 1):
   - Описать проблему
   - Собрать подходящие факты
   - Оценить актуальность и состоятельность
   - Организовать факты
   - Составить пояснения
   - Определить решения
3. Использовать ods/CONSULTANT по следующей схеме (вариант 2):
   - Собрать подходящие факты
   - Определить цели
   - Оценить актуальность и состоятельность

- Определить препятствия
- Произвести учет ресурсов
- Сгенерировать идеи
- Изложить и истолковать идеи
- Предложить решения
- Проверить решения

В чем было основное различие между двумя вариантами? Авторы считали, что вариант 1 ориентирован на выявление базовых причин, переход от причин к объяснениям, а затем к решениям. Напротив, в варианте 2 на каждом этапе внимание обращалось на практические решения. Авторы подытожили их так: «Вариант 1 обращен назад, стремясь к нахождению причин и глубокому пониманию, тогда как вариант 2 обращен вперед в поисках практических решений».

Объектами исследования были практикующие профессионалы примерно одного уровня из аудиторской фирмы Big Eight. Среди заданий были бизнес-задача и задача общественных отношений. Творческий показатель результата решения задачи оценивался комиссией, состоявшей из экспертов, которым было предложено воспользоваться своими субъективными критериями творчества. Предполагалось, что эксперты оценят результаты в соответствии с новизной и пригодностью предлагаемого решения задачи. В качестве экспертов привлекли таких людей, как президенты компаний, консультанты по менеджменту, университетские профессора и директора государственных финансовых агентств.

Какими оказались результаты? Совершенно неожиданными и отчасти тревожными. По системе оценок, в которой чем меньше число, тем выше творчество, результаты были такими:

- Вариант 1: 8,0
- Без программ: 12,7
- Вариант 2: 16,8

Таким образом, существенным оказалось различие между двумя вариантами применения программы (статистически значимым авторы сочли только его), а творческая активность тех, кто вообще не пользовался программой, оказалась в промежутке между двумя вариантами применения программы. Авторы назвали результаты «интригующими», заметив, что «предполагали более творческий результат обоих программных вариантов по сравнению с отсутствием программной поддержки», и подвели итог: «Разницу в результатах двух программных подходов нужно учесть разработчикам СППП... Программное окружение может как способствовать, так и... препятствовать получению нужного результата».

Авторы также зарегистрировали разницу в отношении пользователей к двум вариантам применения программного инструмента. Первый вариант оценивался ими как более увлекательный и полезный, но более трудный в работе.

Кроме того, авторы, не учитывавшие при оценке творчества время, потраченное объектами исследования, отметили, что больше всего времени пользователи потратили в варианте 1.

Авторы не высказались по этому поводу, но из результатов можно сделать вывод, что более творческие решения могут потребовать больше времени, чем менее творческие, но процесс их разработки доставляет участникам больше удовольствия.

Второе исследование

Во втором из упомянутых исследований [Durand 1992] был проведен аналогичный эксперимент. Областью применения творчества была поддержка принятия решений, а коммерческим программным продуктом в этом случае стал Idea Generator. (Утверждается, что этот пакет усиливает возможности мозговой атаки, помогает генерировать альтернативы, откладывать оценку альтернатив до накопления достаточного их числа и проводить выбор среди альтернатив.)

Рассчитывая преодолеть некоторые ограничения первого исследования, выбрали 88 объектов (обладателей MBA с примерно двумя годами опыта работы), предварительно обследовав их на наличие творческих способностей, назначили каждому для изучения по два небольших конкретных случая и отобрали группу независимых судей для оценки результатов по полному набору критериев, лишь часть из которых относилась к творчеству. (Ни судьи, ни объекты не знали, что объектом эксперимента было творчество.)

Объекты работали над конкретными примерами с помощью программы или без нее.

Какими же оказались результаты второго исследования? Для 28 методов оценки, из которых только пять были связаны с творчеством, *использовавшие программное обеспечение объекты показали худшие результаты во всех до единой категориях*. Повторю еще раз: программа не только не способствовала росту творчества (или каких-либо других характеристик решения задачи), но фактически оказалась вредна.

Чем объясняются такие неожиданные результаты? Может быть, объекты неправильно использовали программу? Нет, исследователи получили данные о том, что программа применялась для решения тех задач, кото-

рые она поддерживает (генерация большого числа идей, описываемых подробно и глубоко).

Были ли отклонения, связанные с различными природными уровнями творческих способностей? Нет, программа снижала эффективность работы всех без разбора!

В чем же тогда дело? Назвав результаты «четкими и неожиданными», авторы признаются, что «озадачены». Обсуждая результаты, они отмечают, что «решения, полученные с помощью программ, были оценены как менее творческие». Единственно возможным выводом, по их мнению, будет «необходимость дальше разрабатывать программы, стимулирующие творчество, пока они смогут быть полезны в СППР».

В этой книге мы постоянно убеждаемся, что творчество трудно поддается определению, обсуждению и измерению. Даже когда нам удается определить, обсудить и измерить его, проверка уровня творчества дает существенный разброс результатов. (См., например, главу о творческих, интеллектуальных и канцелярских задачах в части I.)

Результаты этих исследований просто повторяют ту тему. Можно, конечно, теоретизировать по поводу усиления способности к творчеству с помощью компьютеров, но при этом ни на минуту нельзя забывать, что мы плохо понимаем, как это нужно делать, и что применение компьютеров для стимуляции творчества может отрицательно, а не положительно, сказаться на полученных результатах.

Такой вывод не слишком обнадеживает. Тем не менее он оказывается верной оценкой современного положения в стимуляции творчества с помощью компьютеров.

Ссылки

DURAND 1992 – «Creativity Software and DSS: Cautionary Findings», *Information and Management* 23, 1992. Douglas E. Durand and Susie H. VanHuss.

ELAM 1990 – «Can Software Influence Creativity?» *Information Systems Research,* March 1990; Joyce J. Elam and Melissa Mead.

THIERAUF 1993 – *Creative Computer Software for Strategic Thinking and Decision Making: A Guide for Senior Management and MIS Professionals,* Quarum Books; 1993. Robert J. Thierauf.

Глава 16 | Парадоксы творчества

Рост влияния плановых отделов и планов ведет к ограничению свободы действий рядовых работников. Поэтому обычно оказывается, что с усилением планового начала у большой группы работников остается меньше возможностей действовать соответственно собственному разумению. От них ждут послушания, а не оригинальности. Такое ограничение инициативы гасит искры творчества, столь важные для успеха предприятия, а кроме того, оно плохо влияет на моральное состояние.

Уильям Ньюман

«Administrative Action: The Techniques of Organization and Management», Prentice-Hall, 1951

Творчество, как мы уже видели, это трудноуловимый предмет. Его пытались определить тысячи раз, и каждый раз определение зависело от того, кто его дает. Его результаты многократно описаны, и, тем не менее, трудно сказать точно, когда результат является творческим, а когда – нет. Можно ли научиться творчеству? Многие считают, что можно, остальные с ними не согласны. В этом неуловимом понятии творчества есть что-то сверхъестественное.

В силу всего сказанного нас не должно удивить, что для творчества характерны парадоксы. В литературе встречаются, по крайней мере, два из них:

Парадокс творчества номер 1. Чем больше творческая личность знает о целевом предмете, тем меньше требуется творчество.

Парадокс творчества номер 2. Чтобы мыслить оригинально, нужно хорошо изучить чужие идеи.

Прелесть в том, что, сопоставив эти два парадокса, мы получаем третий:

Парадокс творчества номер 3. Парадоксы творчества номер 1 и 2 не согласуются между собой.

Разберем каждый из парадоксов по очереди.

> Чем больше творческая личность знает о целевом предмете, тем меньше требуется творчество.

Допустим, вам нужно решить очень сложную задачу, но при этом оказывается, что аналогичные задачи вы уже решали много раз. Естественно, при решении этой новой задачи вы изучите свои прошлые решения, стараясь найти те, которые можно приспособить к новой ситуации.

Однако в результате использования старых решений вы можете утратить право называть свои действия творческими. Если принять определение творчества из [Newell 62], которое уже использовалось в данной книге, то творчество должно включать в себя новизну или ценность, использовать нестандартное мышление, быть результатом высокой мотивации и настойчивости и решать неясные или некорректные задачи. Конечно, повторно использованное решение имеет ценность, но его повторение устраняет новизну; то, вчерашнее, нестандартное мышление быстро становится сегодняшним стандартным подходом; высокая мотивация и настойчивость нужны все меньше; и хотя первоначальная задача казалась неясной и некорректно сформулированной, лучшее знакомство с ней порождает ощущение понимания и определенности.

Вероятно, лучше всего об этом было сказано в [Sasso 1989]. «Какую роль играет *творчество* в процессе разработки? С нашей точки зрения, оно не занимает какую-то определенную позицию в некотором процессе. Скорее, мы видим, как творчество пронизывает все виды деятельности... Мера, в которой это происходит, определяется, в основном, взаимодействием между опытом разработчика и пространством разработки: если его опыт включает в себя общее решение, которое очевидным образом может быть адаптировано к текущему пространству разработки, то потребуется очень немного творчества. Если, напротив, условия разработки оказываются совершенно необычными для разработчика, то практически каждый аспект процесса разработки потребует творческого мышления».

Какая ирония! При решении задачи наибольший потенциал для творческого решения оказывается у тех, кто хуже всего в ней разбирается. Очень странно, правда? В самом деле, как это вообще можно понять?

Чтобы мыслить оригинально, нужно хорошо изучить чужие идеи.

Столь же парадоксальная мысль. Неужели знакомство с работой других людей не скажется отрицательно на оригинальности наших трудов?

Те, кто обнаружил этот парадокс, считают, что он возникает из того, что они называют «четырьмя этапами творчества»: подготовка, вынашивание, вдохновение и проверка. Согласно такому представлению о жизненном цикле творчества творческая мысль не возникает в пустоте. Она должна стимулироваться (в период подготовки и вынашивания), прежде чем сможет принести плоды (вдохновение). И именно во время стимуляции, когда тот, кто решает задачу, исследует ее предметную область, поглощаются стимулы, которые впоследствии могут привести в движение творческие силы. Например, [Henle 1962] (цитируется по [Couger 1990]) говорит:

> «По-видимому, творческие идеи возникают у нас только после того, как мы потратим достаточно времени и сил, занимаясь делом, которое как раз больше всего затрудняет их возникновение. Возможно, погружение в предметную область является необходимым условием творческого мышления не только потому, что оно снабжает нас материалом для обдумывания, но и потому, что мы постигаем сложность задачи».

Конечно, не все творческие идеи возникают как результат четырехэтапного процесса. Хорошо известно, что решение может быть результатом счастливого случая («АГА!»). Но даже когда решение возникло в результате внезапного озарения, по некотором размышлении вы приходите к выводу, что какая-то часть мозга занималась вводом и обработкой информации о задаче, даже если снаружи это не было заметно.

Этот парадокс, в отличие от первого, более убедителен. В какой-то мере он отвечает западной этике, согласно которой решение проблемы скорее откроется тому, кто работает над ней, а не бездельнику.

В результате приходим к третьему парадоксу.

Парадоксы творчества номер 1 и 2 не согласуются между собой.

Как может быть, что для творчества нужно погрузиться в предметную область, притом что извлечение из памяти прежних решений – это не творчество?

Возможно, следующие рассуждения помогут разрешить эту дилемму.

Парадокс 1 предлагает ситуацию, в которой решающий задачу уже погрузился в нее и (в какой-то прошлый момент) придумал решение. В процессе получения такого решения решающий миновал стадии новизны, нестандартного мышления и необходимости проявления настойчивости, перенеся задачу в разряд корректно определенных.

Напротив, парадокс 2 предлагает ситуацию, когда решающий задачу находится на ранней стадии творческого процесса ее решения. Подготовка и вынашивание должны предшествовать озарению. Готовых старых решений нет.

Таким образом, можно считать, что разница в количестве. Некоторое ознакомление с задачей необходимо, чтобы запустить процесс творчества, но оно должно быть недолгим, иначе перенесет в другую фазу жизненного цикла творчества.

При таком взгляде третий парадокс, хотя он и интересен как семантическое упражнение, вероятно, не столь важен, как первые два. И хорошо. У нас достаточно проблем с выяснением природы творчества, чтобы дополнительно заниматься противоречивыми парадоксами!

Ссылки

COUGER 1990 – «Ensuring Creative Approaches in Information System Design», *Managerial and Decision Economics,* Vol. 11, pp. 281–295, 1990. J. Daniel Couger.

HENLE 1962 – «The Birth and Death of Ideas», in *Contemporary Approaches to Creative Thinking*, Atherton Press, p. 43, 1962; M. Henle.

NEWELL 1962 – «The Processes of Creative Thinking», in *Contemporary Approaches to Creative Thinking,* Atherton Press, 1962; A. Newell, J. Shaw and H. Simon.

SASSO 1989 – William C. Sasso and Monte McVay. «The Constraints and Assumptions Interpretation of Systems Design: A Descriptive Process Model», submitted to the *Journal of Systems and Software* in late 1989 but never published.

Глава 17 | Так было всегда

Философам-романтикам было неинтересно разбирать вселенную как механизм, и анализировать ее мельчайшие частицы. Нет, они стремились созерцать, понимать, толковать, чувствовать и прозревать весь мир в его скрытом значении, как мы воспринимаем стихи или картины.

Бо Дальбом и Ларс Матиассен

«Struggling with Quality, The Philosophy of Developing Computer Systems», Academic Press, 1992

Мы, программисты, склонны считать, что задачи, с которыми мы сталкиваемся, столь же свежи и новы, как наша профессия.

Нам нравится в это верить. С удовольствием применяя для решения задач всякие замечательные новые технологии, приятно полагать, что задачи, которые мы решаем, такие же новые.

Некоторые из них, безусловно, новые. В докомпьютерную эпоху исследование космоса и автоматическое управление самолетом были всего лишь мечтами, но не реальными задачами, допускающими какое-либо осмысленное решение.

Однако большинство решаемых сегодня задач не новы. Информационные, финансовые, производственные системы были возможны и существовали задолго до появления компьютеров. Сейчас они стали гораздо эффективнее, но сами по себе эти задачи не новы.

В области применения компьютеров нередки попытки использовать прежний опыт. Изучая богатое идеями прошлое других дисциплин, люди стараются отыскать идеи, которые могут оказаться полезными в новую эру компьютеров, и это достойные попытки.

Но было в этом нечто странное. Мы пытались заимствовать идеи из области решений других дисциплин, не обращая внимания на то, что узнали об области задач.

Что я имею в виду?

С появлением программного инжиниринга неоднократно делались попытки применить в программировании традиционные инженерные методы на основе аналогии. Например, мы изучали сборочное производство и статистический контроль качества. Однако применение тех же методов в программировании столкнулось с проблемами. Применять аналогичные решения мешают некоторые фундаментальные особенности программирования. Из-за уникальности, невесомости, безразмерности и почти нулевой стоимости производства ПО многие идеи плохо приживаются на этой почве.

Возможно, мы смотрели на аналогии не с той стороны. Перечисленные выше вопросы касаются методов решения. А что если попробовать выяснить, как решали задачи в прошлом? Решение задач, какой бы старой ни была эта дисциплина, не столь богато идеями, как нам того хотелось бы, но все же при решении задач программирования там можно найти некоторые полезные аналогии.

Жизненный цикл как универсальный алгоритм решения задач

Возьмем, к примеру, идею жизненного цикла программы. Представить жизненный цикл можно по-разному, но большинство согласится с тем, что он имеет примерно такой вид:

1. Постановка задачи.
2. Определение требований задачи.
3. Разработка решения задачи.
4. Реализация проекта.
5. Проверка реализации.
6. Применение реализованного продукта.

Достаточно начитанным программистам, которым, возможно, уже доводилось спорить о жизненном цикле, такое изложение может показаться упрощенным и малоинтересным. Но рассмотрим этот жизненный цикл с более традиционной точки зрения на решение задач.

Перечисленные этапы являются, в сущности, универсальным алгоритмом решения задач. Ни один из них не присущ исключительно программированию. В классических книгах по решению задач, например Пойа (математика) или Александера (архитектура), вы найдете нечто очень похожее на этот жизненный цикл. И это еще не все. Такой же подход применяют писатели и художники. А на встрече в клубе деловых женщин мне сказали, что правильный подход к открытию нового бизнеса напоминает этот жизненный цикл!

Рассматривая жизненный цикл как аналогию, заимствованную из общего опыта решения задач, программисты избежали бы множества споров по поводу того, полезен жизненный цикл или нет. Конечно, полезен: мы, программисты, не собираемся создавать заново теорию решения задач. В своих бесконечных спорах об уместности жизненного цикла мы часто упускали из виду то, что его основная схема допускает множество вариаций. Например, спиральная модель и прототипирование – просто варианты базового жизненного цикла. Возможность итераций внутри этапов жизненного цикла может вызывать сомнения только у самых педантичных его приверженцев.

Жизненный цикл как универсальный алгоритм решения задач – не единственное, что мы можем взять для себя, рассматривая решение задач в целом вместо конкретных областей решений. В этой главе нас как раз интересует нечто другое.

Эта книга – размышление о двух подходах к созданию программного обеспечения. Один из них ориентируется на управление, контроль, формализм и дисциплину. Другой – на личное мастерство, творчество, неформальность и свободу. Выбор того или иного подхода может сильно повлиять как на процесс создания программного продукта, так и на его результат. Предположение, высказанное в начале книги и разными путями подтверждаемое на всем ее протяжении, заключается в том, что правильный подход к созданию программ должен в какой-то мере сочетать оба эти принципа. Задачей книги было подчеркнуть значение второго из них, поскольку силы, управляющие миром программирования, фактически сосредоточились исключительно на первом, в результате мы теряем нечто ценное.

Но постойте. Эта глава должна была показать, что полезен бывает исторический опыт решения задач. Может, это справедливо и в отношении дихотомии подходов? Не поискать ли нам что-либо, касающееся дилеммы римляне/греки (руководители/технические специалисты), в старых информационных ресурсах?

Механистический/романтический, жесткий/мягкий, рациональный/эмпирический

Ответ, как и ожидалось, положительный. Этой дихотомии много веков. Очень удачно она описана в [Dahlbom 1992].

Дальбом обозначает эти два очень разных подхода как механистический, ориентированный на представление, формализацию, план, порядок и контроль, и романтический, для которого характерны индивидуальная художественная выразительность, гений и творчество. «Основное различие между организмами и машинами, – пишет Дальбом, – состоит в романтической реакции на механистическое видение мира, в желании защитить природу и все естественное от машин и всего искусственного».

И продолжает: «Философам-романтикам было неинтересно разбирать вселенную как механизм и анализировать ее мельчайшие частицы. Нет, они стремились созерцать, понимать, толковать, чувствовать и прозревать весь мир в его скрытом значении, как мы воспринимаем стихи или картины».

Дальбом назвал эту дихотомию «классическим противоречием между управлением и творчеством». И подчеркнул ее значение в так называемом споре между «жесткими» и «мягкими» методами. Смысл этих двух терминов здесь несколько отличается от интуитивного. «Жесткий» означает объективный, алгоритмический, основанный на фактах, представляющий мир точным и правильным. В исторической перспективе это «механистический». «Мягкий» означает нечеткий, состоящий из идей и понятий, допускающих разные толкования, подчеркивающий культурные различия. В исторической перспективе это «романтический». Жесткие предметы легко изучать: их можно запомнить, можно проверять их знание методом выбора правильного варианта или ответа «да/нет». «Мягкие» предметы изучать трудно: запомнить невозможно, проверка выполняется с помощью обсуждения или сочинения.

Учебные заведения, особенно в Соединенных Штатах, предпочитают жесткие предметы. О них даже говорят: «Жесткое вытесняет мягкое». (То есть те, кто поддерживает жесткие предметы, склонны отказывать тем, кто поддерживает мягкие. К примеру, нередко по этому принципу ведутся битвы за штатные должности.) Преподавать и проверять предмет, представляющий собой набор фактов, достаточно приятно (и легко!). А преподавать предмет, представляющий собой набор идей, которые сегодня могут значительно отличаться от вчерашних или завтрашних, достаточно хлопотно (и непросто!).

Обычно студенты чувствуют себя увереннее с «жесткими» предметами. С младших классов нас приучают к мысли о том, что образование состоит

в накоплении фактов. Если учащийся, продвигаясь по образовательной лестнице, не сообразит, что у зрелой личности образ мышления, основанный на фактах, дополняется концепциями, основанными на идеях, его ждут большие сложности при изучении «мягких» предметов. Это касается не только таких традиционно «мягких» предметов, как философия или социология, но и ряда дисциплин, связанных с программированием. Например, явно «мягкими» являются темы, относящиеся к менеджменту. При отсутствии готовой формулы лучшего решения для всех задач «мягким» предметом оказывается и программный инжиниринг. (Некоторые специалисты по компьютерным наукам пытаются сделать его «жестким», преподавая формальные методы создания программ, но автор этой книги склонен считать, что преподавать только формальные методы по крайней мере преждевременно, поскольку их превосходство над неформальными пока не продемонстрировано никакими оценочными исследованиями. Вдобавок этот подход столь же вдохновляет учащихся, как преподавание истории в виде списка исторических дат – верный способ отбить всякий вкус к предмету!)

В книге Дальбома идея противопоставления механистического и романтического, превратившихся в жесткое и мягкое, делает еще один эволюционный виток. Он проводит различие между «рационализмом» (теорией, согласно которой в основе достоверности знания лежит рассуждение) и «эмпиризмом» (учением о непосредственном получении знания через чувственное восприятие). Это различие он развивает в противоположность между конструированием систем, которое, по его мнению, «принадлежит к рационалистической традиции», и решением задач, которое «выражает более эмпирические идеи». «Главная проблема в конструировании систем, – говорит он, – сложность, тогда как главная проблема методов решения задач – неопределенность. Отсюда правило: в случае, когда сложность задачи высока, а неопределенность низка, следует предпочесть метод конструирования систем, а в случае высокой неопределенности и низкой сложности следует предпочесть метод решения задач».

Как бы то ни было, во всем этом спектре дихотомий Дальбом видит любопытный проблеск надежды. «Резкий антагонизм между рационалистами и эмпириками со временем смягчился, – говорит он. – В этом веке наука пришла к несколько неопределенной, но здоровой комбинации обоих методов. Разновидность индукционизма, требующая беспристрастности наблюдения, была признана неосуществимой и уступила место проверке теоретических гипотез. С помощью индукции вы заключаете, какие эмпирические последствия должны иметь ваши теории, а потом проверяете, таковы ли эти последствия, получая данные и упорядочивая их с помощью индукции. Хотелось бы видеть в методах решения задач [только] такую полезную смесь дедукции и индукции».

Что все это значит? То, что дихотомия римляне/греки, руководитель/технический специалист, механистический/романтический, жесткий/мягкий, рациональный/эмпирический – или как вы предпочитаете ее называть – уходя корнями в древность, не является уникальной особенностью программирования. Напомним, что Дальбом назвал ее «классическим противоречием между управлением и творчеством».

Так было всегда. Едва ли нам, программистам, хватит жизни на то, чтобы соединить дихотомии, покончив с противоречием, и тем самым решить задачу, стоявшую еще перед нашими собратьями в гораздо более древних отраслях. Но, по мнению Дальбома, не исключено, что «резкий антагонизм» со временем сможет «смягчиться», как это было в тех более старых областях. В частности, я надеюсь, что эта книга, весьма симпатизирующая творчеству в программировании и в то же время признающая необходимость дисциплины и формализма, поможет приблизить это смягчение, жизненно необходимое в нашем деле.

Ссылки

Dahlbom 1992—«Struggling with Quality, The Philosophy of Developing Computer Systems», pp. 16–17, 52–55, 94–95, 142–143, Academic Press, 1992; Bo Dahlbom and Lars Mathiassen.

Часть IV

Итоги и выводы

Введение
18. Синергическое заключение
19. Еще несколько выводов

Введение

Мы учимся новому, когда воспринимаем информацию, которая вступает в конфликт с нашими моделями. Гармония порождает благодушие.

Моше Рубинштейн

«Tools for Thinking and Problem-Solving», Prentice-Hall, 1986

Рецензент одной из моих старых книг высказался довольно точно. Дочитав рукопись до конца, он воскликнул: «Напомните мне, из-за чего возник спор? Здесь масса разных мыслей, и автору неплохо бы подвести в конце итоги».

Последовав совету этого рецензента, я с удовлетворением написал заключительный раздел для той книги. Не только читатель, но и я сам получил более четкую картину того, куда привела нас книга.

Поэтому то же самое я пожелал сделать и с этой книгой. В результате появились две главы части IV.

Глава 18 | Синергическое заключение

Идеи правят миром, так же как деньги, власть и секс.

Джон Форбс Нэш мл., математик
и лауреат Нобелевской премии по экономике,
из его биографии «A Beautiful Mind»,
написанной Сильвией Наса

Пытаясь создать целостный образ из различных граней творчества в программировании, освещаемых в этой книге, я вдруг заметил нечто удивительное. Я начал ощущать синергию: образ стал превращаться в нечто большее, чем просто сумма отдельных частей книги.

Когда я пересматривал материал, у меня в голове зазвучало несколько тем. Вот тема Джеффа Оффута, который запирает свою лабораторию, забывает все формальные методы, которым только что учил студентов, и выбирает вместо этого творческие решения *ad hoc*.

Вот тема Дэвида Парнаса, признающегося в том, что нисходящая разработка на самом деле просто невозможна, и призывающего «подделывать» итоговую документацию.

Вот тема Билла Кертиса, предполагающего найти в разработке сбалансированное и управляемое развитие и в конце концов приходящего к противоположному выводу о том, что разработка является оппортунистическим процессом.

Вот тема Дугласа Кинга, цитирующего слова Херба Саймона о том, что решать хорошо понятые задачи с помощью формальных методов – совсем

не то же самое, что решать малопонятные задачи с помощью слабо структурированных эвристических методов.

Вот тема Брюса Блюма, сообщившего нам, что оптимизировать процесс ради удовольствия можно было тогда, когда задачи были маленькими, а стремление к качеству основывалось на личных представлениях, и что оптимизировать процесс для совместимости с поэтапной моделью допустимо только при наличии твердых и устойчивых требований.

Среди всех этих тем я начинаю различать одну общую. По некотором размышлении я готов описать ее следующим образом.

В нашем академическом мире создание программного обеспечения воспринимают как некий монолит. Мы придумываем и пропагандируем методы, языки, компьютеры и инструменты, которые должны быть пригодны на все случаи жизни. От каждой новой методологии мы ждем решения задач всех типов. Мы создаем модель проверки процесса, ожидая, что она будет применима для всех программных проектов.

Думаю, все эти темы дают основания для вывода о том, что поиск универсального подхода – это ошибка. Даже еще хуже. Это ОШИБКА! Нам необходимо начать учитывать в программных разработках их индивидуальную природу, соответственно управляя разработкой и решая задачу.

Эта мысль не была для меня совершенно новой. Ранее я уже несколько раз исследовал зависимость методов решения от предметной области и понял, что разные области требуют различных методов – в значительно большей степени, чем мы это себе представляем. Но все эти мысли хранились в моей голове отдельно как тема другой серии очерков и статей, и я не сразу заметил их близость к проблеме творчества в программировании.

Я также провел некоторое исследование таких аспектов программных проектов, как размер и критичность. Например, в приложениях моей книги «Building Quality Software» (Разработка качественного программного обеспечения), Prentice-Hall приведены рекомендации по методам создания качественных решений исходя из размера и критичности конкретного проекта.

Однако из этих творческих тем возникает целый новый аспект – «новизна решаемой задачи». Один умный автор за другим, каждый по-своему, говорят о том, что «чем более новаторские задачи встают перед нами, тем менее эффективны формальные методы» и что «строгие методы нужно хотя бы дополнять творческими, когда задачи требуют больших умственных усилий».

Я стал искать какой-нибудь важный критерий для представления программных проектов. Ясно, скажем, что размер проекта играет важную роль при выборе технологий для решения. Точно так же ясно, что кри-

тичность проекта играет такую же роль. Короче говоря, методы, применяемые для решения задач в программировании, зависят от множества аспектов.

Нужно учесть, что мои поиски начались как раз в то время, когда институт SEI стал пропагандировать модель зрелости процесса (CMM) и пятиуровневую систему оценки зрелости управления, которые предлагались для всех программных проектов без разбора. В это же самое время для применения во всех проектах рекламировались все новые формальные методы и методологии, предлагаемые компьютерными науками или коммерческими организациями.

Думаю, что через несколько лет выстраданные догадки, о которых я сейчас пишу, будут восприниматься как нечто очевидное. Но в ту эпоху, когда я извлекал эти идеи из *комментариев* своих коллег, они, конечно, выглядели радикальными.

Что я пытаюсь сказать? То, что при оценке пригодности каждого нового метода разработки программного обеспечения должны учитываться несколько аспектов проекта:

- Характер и требования предметной области
- Масштабы проекта
- Критичность решения
- Новизна задачи

Возможно, есть и другие аспекты – вопрос не закрыт. Например, кандидатами на включение в список могут быть знания/стиль руководителей проекта или качество/возможности технических сотрудников.

Интересно отметить, что эти кандидаты возникают при выборе методов, ориентированных на организацию, а не на проект. Эта мысль интересно разработана в [Constantine 1993], где предлагаются организационные парадигмы для разных типов проектов:

- Замкнутый (традиционная иерархическая организация – Константин говорит, что такой подход хорош для «стандартных, тактических проектов»)
- Случайный (на основе независимой новаторской инициативы – Константин считает этот подход лучшим для проектов, требующих «творческого прорыва»)
- Открытый (адаптивный, совместный – лучше всего для «решения сложных задач»)
- Синхронный (эффективное гармоничное регулирование – лучше всего для «повторяющегося, критического исполнения»)

[Hyman 1993] содержит прекрасные конкретные случаи применения некоторых из этих концепций.

Но задержимся на некоторое время на тех подходах к проекту, которые учитывают его особенности.

Попробуем оценить разные темы из части I книги по каждому из следующих аспектов.

Дисциплина. Строгая дисциплина необходима для сверхкрупных или сверхкритичных проектов. В других проектах ее значение меньше.

Формальные методы. Формализм уместен в малых проектах (возможно, что и в крупных, но нужны дополнительные исследования); мы не знаем пока, важен ли он для критичных проектов, хотя многие в этом уверены; главное, что он применим только в проектах, новизна задач в которых невелика. Для сложных проектов более эффективными оказываются эвристические методы. (К примеру, Реттиг и Саймонс придерживаются мнения, что для сложных задач лучше всего подходит «итеративная стратегия», включающая в себя методы «разделяй и властвуй», спиральный жизненный цикл, объектную технологию, быстрое создание прототипов и инкрементное тестирование [Rettig and Simons 1993].)

Оптимизирующие решения. Оптимизация возможна только в малых проектах. Она относительно независима от критичности проекта (хотелось бы получать оптимальные решения для критичных проектов, но это невозможно) или с новизной задач.

Количественные рассуждения. Несмотря на то что количественные методы желательны, по всей видимости, они применимы только в относительно небольших и не отличающихся новизной проектах.

Процесс. Вероятно, процесс должен идти рука об руку с дисциплиной. То есть, чем больше потребность в дисциплине, тем большая необходимость определить процесс, для которого будет установлена дисциплина. См. выше обсуждение дисциплины.

Интеллектуальные, творческие задачи. Потребность в интеллектуальных и творческих методах пропорциональна размеру проекта, критичности решения и в особенности новизне задачи.

Увлекательность. Значение увлекательности, видимо, обратно значению дисциплины. То есть чаще всего оно ощущается в небольших проектах, которые не критичны, но содержат задачи, отличающиеся хоть какой-то степенью новизны.

Тут у нас возникает проблема. Все перечисленное потребовало многих слов для описания. А слова могут затуманить простоту основной идеи. Я попытался изобразить взаимодействие этих аспектов и проблем творче-

ства простым графиком, но мне это не удалось. Поэтому, прежде чем идти дальше, хочу еще раз подчеркнуть основную мысль:

Подход с одной меркой ко всем задачам и решениям в программировании мешает заметить некоторые важные вещи. Эти важные вещи помогают определить, когда определенные методы решения действуют хорошо, а когда – плохо. Такие разрекламированные подходы, как дисциплина, формализм и установленный процесс, предлагаемые для всех проектов без учета их специфики, могут применяться только для определенной группы задач. Необходимо начать обсуждение того, в каких случаях применим (а еще важнее – когда не применим) каждый подход. Данный раздел может послужить началом такой дискуссии.

В области человеческих отношений, особенно когда речь заходит о гражданских правах, часто «восхваляются различия» между людьми. Имеется в виду, что расовые, половые, религиозные и некоторые другие важные различия должны уважаться и изучаться, а не вызывать страх или отторжение. Пора и в компьютерном сообществе начать уважать различия. И обсуждавшиеся выше аспекты – предметная область, размер проекта, критичность решения, оригинальность задачи – устанавливают те типы различий, которые в области компьютеров и программирования нужно начать понимать – и уважать.

Ссылки

CONSTANTINE 1993—«Work Organization: Paradigms for Project Management and Organization», *Communications of the ACM*, Oct. 1993; Larry L. Constantine.

HYMAN 1993—«Creative Chaos in High-Performance Teams: An Experience Report», *Communication of the ACM*, Oct. 1993; Risa B. Hyman.

RETTIG AND SIMONS 1993—«A Project Planning and Development Process for Small Teams», *Communications of the ACM*, Oct. 1993; Marc Rettig and Gary Simons.

Глава 19 | Еще несколько выводов

Выдающиеся ученые прекрасно справляются с неопределенностью. Они достаточно верят в теорию, чтобы идти вперед, и достаточно сомневаются в ней, чтобы, заметив ошибки и заблуждения, создать вместо нее новую. Если слишком верить, не увидишь изъянов; если слишком сомневаться, не тронешься с места. Нужен тонкий баланс.

Ричард Хемминг

«You and Your Research»,
(запись выступления на семинаре Bell Communication Research Colloquium Seminar 7 марта 1986 г.)

Пожалуй, уважение к различиям между многочисленными аспектами программных проектов – это главная мысль, которую я хотел бы донести до читателей этой книги, но есть и другие.

Прежде всего, создание этой книги, если вспомните, становилось моей личной одиссеей, по мере того как моей взгляд на программирование как на творческое искусство изменялся в пользу более сбалансированного представления о сочетании дисциплины и творчества. Возможно, по прочтении этой книги с вами произойдет то же самое.

Эта одиссея, как мне хотелось бы верить, дала еще несколько интересных мыслей.

У подходов к созданию программного обеспечения есть важные культурные различия. Какое ни выбери – между Agile-программированием и струк-

турным программированием, между открытым и коммерческим программированием, между греками и римлянами, между технологическим управлением и менеджментом, между конвейером и мастерами-умельцами или, обратившись к прошлому, между механистическим и романтическим или рациональным и эмпирическим – они оказываются вполне реальными, если рассмотреть их влияние на то, *как* мы пишем программы.

Чтобы доказать превосходство одной культуры над другой, данных недостаточно. Однако у краев разделяющей их пропасти встречаются результаты, из которых следует, что в определенных ситуациях одна из сторон оказывается права.

Вероятно, самые интересные в этой книге данные получены в результате пары экспериментов по выяснению характера задач программирования. В одном из этих экспериментов задачи были разделены на творческие, интеллектуальные и канцелярские, и определенно выяснилось, что в программировании примерно 80% задач являются интеллектуальными и только 20% – канцелярскими. Степень творческого характера этих задач определена в другом эксперименте, результаты которого менее убедительны. Экспертные оценки весьма различны, и если одни эксперты назвали творческими лишь 6% задач программирования, то другие повысили эту оценку до 29% (в среднем она составила 16%). Как ни толкуй эти результаты, они явно – количественно – показали, что задачи программирования требуют значительных интеллектуальных усилий.

В каких задачах необходимо творчество? По этому вопросу согласие достигнуто не было, но к таким задачам, несомненно, относятся следующие:

- Перевод бизнес-требований на язык задач, подлежащих творческому решению (сюда входят задачи стратегического планирования и конкурентного преимущества).
- Разрешение конфликтов в постановке задачи, выполняемой несколькими предприятиями.
- Разработка решений для сложных и новых задач.
- Создание необходимого и достаточного комплекта тестовых условий и примеров.
- Крупные усовершенствования программных систем, когда такие усовершенствования не предусмотрены первоначальной конструкцией системы.

Если считать, что при создании программных решений творчество необходимо, какие последствия это имеет для набора кадров? На протяжении всей книги мы встречали подтверждения того, что способность к творчеству есть у каждого. Мы также видели, что согласно исследованиям покойного Дэна Кугера стоимость обучения, позволяющего развить природ-

ные творческие способности, не превышает 0,5% стоимости проекта. Кроме того, мы увидели, что есть много хорошо известных приемов стимулирования творчества и, наконец, что эти приемы могут повысить индивидуальную способность к творчеству примерно на 23%, а уровень творчества в организации, где работают эти отдельные личности, – на 27% (согласно одному неформальному исследованию).

Довольно любопытно было узнать, что программные инструменты для усиления творческих возможностей людей, не связанных с программированием, оказывают смешанный, а порой и отрицательный эффект. Похоже, в данное время мы достигнем большего, применяя творчество в задачах программирования, а не программы в задачах творчества.

Есть еще несколько мыслей, которые (по крайней мере у меня) останутся от связанной с этой книгой одиссеи. Это выразительные цитаты о роли творчества в программировании. Вот некоторые из моих любимых:

> По поводу количественных подходов: «Если достаточно долго пытать числа, они скажут вам все, что вы хотите узнать».
>
> По поводу «нельзя управлять тем, чего не можешь измерить»: «Мы только этим и занимаемся».
>
> О роли дисциплины и процесса: «Мы следовали всем правилам и завалили проект».
>
> О войне между теорией и практикой: «Слабые теории со смехом изгоняются оттуда, где царит практика», «Когда не используются законные выводы науки, в практике случаются застои» и «Если теория сражается с практикой, это явный признак того, что не все в порядке».
>
> Об исследователях, на работы которых никто не ссылается, но которые при этом рассчитывают на правительственные субсидии: «„Королевы на пособии“ в белых халатах».
>
> Об универсальных подходах и методах *ad hoc*: «Универсальные подходы „слабы“ в том смысле, что нет таких задач, которые они решают хорошо; методы *ad hoc* „сильны“ тем, что нацелены на практическую задачу».
>
> О конфликте между творчеством и дисциплиной: «Они составляют „странную парочку“, поскольку оба необходимы».

Последняя цитата замыкает круг. Какими должны быть методы программирования – формальными и дисциплинированными или свободными и творческими? В конечном счете, мысль этой книги ясна. Ответ прост:

«Да!»

Алфавитный указатель

Специальные символы

4GL, 298
5W и H, 264

A

ad hoc, 86, 91, 92
Ada, язык программирования, 51, 293
Adaptive Software Development, 49
Agile, 48, 50, 52, 231
American Airlines, 254, 257

B

Beck and Boehm, 51
BIEGE, 85
Blum, Bruce, 66
Boehm, Barry, 50
Brooks, Fred, 37

C

Capability Maturity Model, 118
Carr, Nicholas, 253
CASE-средства, 38, 53
Cleanroom, методология, 94
CMM, 118, 134
Cockburn, Alistair, 50
Coles, Roland, 259
COMDEX, 110
Communications of the ACM, 131
Computer Science Technology Board of the National Research Council, 66
computer-aided manufacturing, CAM, 283
CopyLeft, 259
Cox, Charles, 248
Creative Commons, 259
Crystal Clear, 48
Curtis, Bill, 46, 270
Cusumano, Michael A., 249
Cutter IT Journal, 89

D

Davis, 38, 39
Dekleva, Sasa, 121
DeMarco, 40
Denning, Peter J., 78, 268
Dynamic Systems Development, 49

E

EDSAC, 293
Elsevier Science, 30
«Empirical Studies of Programmers», 38
Engelbart, Doug, 219
Eolas Technologies, 259
ESP, 273

F

«falutin' index» («индекс сложности»), 56
Feature-Driven Development, 49
Fetzer, James H., 63
Fortune 500, 122
Fraunhofer, 194

G, H

Greenbaum, Joan, 301
Groen, 41
Herblseb, Jim, 134
Humphrey, Watts, 250

I

IBM, компания, 133, 252
IBM 360, 105, 297
Idea Generator, 323
Internet Explorer, 259

J

J. Lyons, компания, 293
Jeffery, 40
Journal of Systems and Software, 131

Judson, 42

K, L

Kraft, Philip, 301
Lammers, Susan, 269
Lampson, Butler, 275
Lindvall, 52

M

McClure, Carma, 37
Microelectronics and Computing Consortium, MCC, 46
Microsoft, 259
Midnight Engineering, журнал, 316
Moad, 36, 39

N

Nestor, John, 269
New York Times, 139
Newman, William H, 325
NICTA, 38, 194

O

ods/CONSULTANT, 321
Open Source, 228, 233, 259
Organization Development Software, 321
OS/2, 133
OS/360, 133

P

Parnas, David L., 37, 74
PeopleSoft, 297
problem-solving, 64

R

Rifkin, Stan, 248
Rombach, H. Dieter, 66

S

SAP, 297
satisficing, 84
Scrum, 49
SEI, 134
Simon, Herbert, 65
Simula, 38
Skowronski, 52
Software Cost Reduction Project, 68, 74
Software Engineering Institute, 38, 194, 201
Soloway, Elliot, 270
Sutherland, 51

T, U

TEDIUM, 232
Tetlock, Philip E., 94
The Discipline of Curiosity, 30
The Wall Street Journal, 130
United Technologies Microelectronics Center, 279

W

Wal-Mart, 254, 257
Washington Post, 94
Williams, 51
Wing, Jeanette, 67

X, Z

Xerox PARC, 181, 194
Zelkowitz, Marvin, 45

А

Александер, Кристофер, 179
Али Мили, 131
альтернативные определения жизненного цикла, 45
анализ предметной области, 93
аэродинамика, 179

Б

Бернерс-Ли, Тим, 260
Блюм, Брюс, 66, 228
Боэм, Барри, 50, 51, 120
Брукс, Фред, 31, 95, 298

В

«В поисках решений», 62
взаимное непонимание между теоретиками и практиками, 214
влияние возраста на творчество, 252

Г

Галилей, 226
Гейтс, Билл, 230, 272
гибкое программирование, 48, 51
гибкость, 30
Горбачев Михаил, 86
голубые карточки, метод, 280

греки и римляне, 245
Гринбаум, Джоан, 301
Гроэн, Дженни, 30

Д

Дегрейс, Питер, 244
Дейкстра, Эдсгер, 63, 69, 188
деквалификация программистов, 301
Деклева, Саса, 121
Демарко, Том, 40, 139
Деннинг, Питер, 78, 268
Джадсон, Гораций Фриленд, 62, 176, 200, 226
Джесси Хелмс, сенатор, 94
дисциплина, 30, 40, 58
достаточно хорошее ПО, 89

Ж, З

жизненный цикл, 43
заговор менеджмента по деквалификации программистов, 303
Заде, Лотфи А., 98
закон планирования Грешема, 80
Зелковиц, Марвин, 45
Земанек, Хайнц, 184

И

изучение разработки путем опросов, 272
имитация нисходящей разработки, 46
индекс сложности, 56
институт программирования (SEI), 38, 118
институт Фраунхофера, 38
«Инструменты мышления и решения задач», 84
интерфейс пользователя, 181, 193
интуиция как метод принятия решений, 109, 121
информационные системы, 218
исследования, 217
исследовательская лаборатория ВМФ, 68
исторические даты в программировании, 292
«ИТ ничего не дают», 253

К

какой длины должны быть рассказ или выступление, 86
Карр, Николас, 244, 253
качества выдающихся разработчиков, 269
качество и количество, 98
качество, измерение, 99
Кеннеди, Ной, 118, 176
Кертис, Билл, 46, 120, 270
Килдалл, Гэри, 272, 275
Кин, Питер, 190
Кнут, Дональд, 184
когнитивные модели, 66
Кокберн, Элистер, 50
Кокс, Чарльз, 244, 248
компиляторы, 181
компьютерные науки и чистая математика, 78, 92
консорциум микроэлектроники и вычислительной техники, 46
контроль, 40
конфликт между формальными и неформальными методами, 80
Коулз, Роланд, 259
Кохен, Манфред, 189
кратковременная и долговременная память, 274
Крафт, Филип, 301
Кугер, Дэниел, 241, 263
Кузумано, Майкл, 249, 262

Л

лаборатория программирования центра НАСА в Годдарде, 38
Лавлейс, Ада, 293
Ледерберг, Джошуа, 62, 72
Ледерер, 110
Лэммерс, Сьюзен, 269
Лэмпсон, Батлер, 275, 277

М

Маклюр, Карма, 37
манифест Agile, 49
математика и компьютерные науки, 103
местный патриотизм, 204
методы
- 5W и H, 281
- динамической разработки систем, 49
- «манипулирующих глаголов», 265
- последовательных абстракций, 280
- проб и ошибок, 71

метрики в программировании, 102, 196
микроадминистрирование, 55

Миллз, Гарлан, 188
моделирование и эмулирование, 195
модель зрелости процесса создания ПО, 118
оценка результатов применения, 134
модульность, 137
можно ли научиться творчеству, 315
мозговая атака, 280

Н

наборы примитивов у разработчиков, 276
НАСА, 38
«Науки об искусственном», 67, 88
научное программирование, 102
научный метод исследований, 38, 108
Национальный научно-исследовательский совет, техническая комиссия по компьютерным наукам, 66
несоответствие сложности задач и решений, 57
Нестор, Джон, 269
неэгоистичное программирование, 33
нисходящий метод разработки, 46, 93, 94
новые технологии и деквалификация программистов, 301
Ньюмен, Уильям, 325
Ньютон, 52

О

обращение задачи, метод, 281
объектно-ориентированное программирование, ООП, 298
окончательное решение проблемы продуктивности в разработке ПО, 95
олдувайский императив, 248
описание разработки блоков, 44
определение творчества, 326
оптимальное решение, 84
оптимизация, 84, 88
организационное творчество, 310
органы государственной власти и программирование, 64
ориентированность на достижение цели, 215
особенности личности, способствующие индивидуальному творчеству, 310
отношение забава/скука, 229
оценка пригодности новых методов разработки программного обеспечения, 339
оценка сложности мышления и способности действовать в условиях неопределенности, 94

П

парадоксы творчества, 326
Парнас, Дэвид, 31, 37, 46, 74, 76, 95, 189, 275
патентное право, 259
Пейдж, Джон, 272, 276
Плогер, 56
Пойа, Дьердь, 265
полевой анализ действующих факторов, 281
пошаговое согласие, 236
правила контролируемой разработки, 47
практика
опережение теории, 191
премия «Золотого руна», 76
преподавание формальных методов студентам, 69
приемлемое решение, 85
применение компьютеров для усиления творческих возможностей, 320
принцип «лучшее – враг хорошего», 85
принципы сильной связности и слабого сцепления, 267
принятие желаемого за действительное, метод, 281
программирование больших задач, 194
программная физика, 102
продуктивность разработчиков, 40
проект А-7, 68, 74
проект по сокращению стоимости разработки ПО, 68, 74
протокольный анализ, 270
процесс и продукт, 119
«Психологический портрет Горбачева», 94
Путнэм, Ларри, 134

Р

разумная достаточность, 85, 88, 126
решение задач, предметная область, 64
Рифкин, Стэн, 244, 248
Рой, Растом, 190
Ромбах, Г. Дитер, 66
Рубинштейн, Моше Ф., 26, 84, 336

С

Саймон, Герберт, 65, 80, 180
семь плюс-минус два, 274
сильные и слабые методы решения задач, 94
Симони, Чарльз, 272
синдром латыни, 106
системный анализ, 93
системы обеспечения конкурентного преимущества, 253
системы оценок и поощрений, влияние на творчество, 312
системы поддержки принятия решений (СППР), 321
Сковронский, 52
скрам, 49
Советский Союз, 86
Солоуэй, Эллиот, 270
сопровождение ПО, 44, 186, 193
социальные процессы, 63
социология программных проектов, 50
Сталь, Лесли, 244
стандарты документов, 44
стереотип программиста, 39
Стивенс, Уоллес, 315
стоимость сопровождения ПО, 187
странная парочка, 58
стратегические информационные системы (SIS), 253
структурное программирование, 107, 139
структурные методы, 127

Т

творческие личности, 311, 315
творчество и познание, 320
творчество, определения, 263
термодинамика, 180
тест Роршаха, 317
тестирование, 94
Тетлок, Филип, 94
техническая комиссия по компьютерным наукам, 66
техническая подготовка руководителей, 251
трапециевидная призма, 67
Трумэн, Гарри, 73

У

удовольствие от работы с компьютерами, 227
Уинг, Жанет, 67
Университет Колорадо, 241
управление документами, 43
управление и новаторство, 250
уровни процесса SEI, 249

Ф

фабричные трубы, 31
Фетцер, Джеймс, 63
Филипс, Дуэйн, 235
философия конвейера, 33
Фома Аквинский, 52
Форбс Нэш мл., Джон, 337
формализм, 78
формальные методы, 62, 66, 78, 80, 107, 267, 340
  и эвристики, 70
Фрош, Роберт, 124
функционально-ориентированная разработка, 49

Х

Хамфри, Уоттс, 120, 241, 250
Хемминг, Ричард, 244, 343
Херблсеб, Джим, 134
Хетцел, Билл, 103
Хисао Ямада, 200
Холстед, Мюррей, 102

Ц

Центр исследований творчества Университета Колорадо, 241

Э

эвристики, 62
Эдисон, Томас, 252, 258
экстремальное программирование, 48
эмпирические исследования программирования, 273
Энгельбарт, Дуг, 219
этические нормы, 202
эффективность программистов, 41

Я

языки четвертого поколения, 53, 177

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 978-5-93286-152-3, название «Креативное программирование 2.0» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.